

DONALD R. ANTONELLI
DAVID T. TERRY
MELVIN KRAUS
WILLIAM I. SOLOMON*
GREGORY E. MONTONE
RONALD J. SHORE
DONALD E. STOUT
ALAN E. SCHIAVELLI
JAMES N. DRESSER
CARL I. BRUNDIDGE*
PAUL J. SKWIERAWSKI*

RANDALL S. SVIHLA
DAVID S. LEE*
ROBERT M. BAUER
DEMETRA J. MILLS
HUNG H. BUI*

*ADMITTED OTHER THAN VA

LAW OFFICES
ANTONELLI, TERRY, STOUT & KRAUS, LLP

SUITE 1800
1300 NORTH SEVENTEENTH STREET
ARLINGTON, VIRGINIA 22209

September 27, 1999

OF COUNSEL
DALE CURTIS HOGUE, SR.
CHITTARANJAN N. NIRMEL, PHD*

PATENT AGENT
LARRY N. ANAGNOS

TELEPHONE
(703) 312-6600

FACSIMILE
(703) 312-6666

E-MAIL
email@antonelli.com

09/27/99
JCS98 U.S. PTO

JCS98 U.S. PTO
09/405089
09/27/99

Honorable Commissioner of
Patents and Trademarks
Washington, D.C. 20231

Attorney Docket Number: 520.37631X00
Customer Number 020457

Sir:

Attached please find the application papers of Shigekazu INOHARA, Shinji FUJIWARA, Yoshimasa MASUOKA, Nobutoshi SAGAWA, covering new and useful improvements in METHOD FOR OPTIMIZING REMOTE PROCEDURE CALL (RPC) AND PROGRAM EXECUTION METHOD BY USE OF THE OPTIMIZED RPC, comprising:

Specification, Thirty-Two (32) Claims and Abstract of
the Disclosure (62 pages in the Japanese language)

Eighteen (18) Sheets of Drawings Showing Figures 1-21

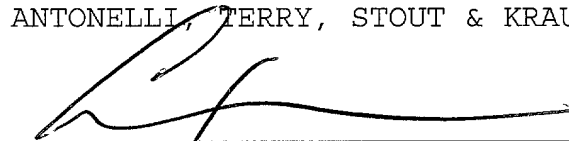
U.S. Government Filing Fee of \$976.00

Change of Correspondence Address

Please charge any shortages in the fees or credit any overpayments thereof to the deposit account of Antonelli, Terry, Stout & Kraus, LLP Account No. 01-2135 (520.37631X00).

Respectfully submitted,

ANTONELLI, TERRY, STOUT & KRAUS, LLP



Carl I. Brundidge
Registration No. 29,621

CIB/nac
Attachments

09/405089 "09/27/99"

明細書
(SPECIFICATION)

発明の名称

(TITLE OF THE INVENTION)

遠隔手続き呼び出し最適化方法とこれを用いたプログラム実行方法
(Method for Optimizing Remote Procedure Call (RPC) and Program Execution
Method by Use of the Optimized RPC)

発明の背景

(BACKGROUND OF THE INVENTION)

発明の分野

(FIELD OF THE INVENTION)

本発明は、計算機システムに関し、特にネットワークによって接続された複数の計算機または処理装置の間で行う遠隔手続き呼び出しの最適化方法及びこれを用いたプログラム実行方法に関する。

従来技術

(DESCRIPTION OF THE PRIOR ART)

コンピュータネットワークによって結合された2つ以上の計算機からなる分散計算システムが広く用いられている。これらの計算機の上で実行状態にあるプログラムまたはプログラム部品（以下、オブジェクト）が2つ以上動作したり、単一のコンピュータ内でオブジェクトが2つ以上動作する場合のオブジェクト間の通信にはいくつかのモデルが存在する。代表的なモデルは、データグラム通信（通信データをパケットとよぶ転送単位ごとに送信・受信するモデル）、ストリーム通信（連続する通信データを一列の流れとし、送信・受信を任意の長さで行うモデル）、分散共有メモリ（特定のメモリアドレスへの更新と参照を、通信データの送信と受信に対応させるモデル）、そして遠隔手続き呼び出し（手続きの呼び出しを

通信に対応させるモデル)である。手続きは、関数、サブルーチン等とも呼ばれる。以下の記述では、これらはすべて同じ意味で用いる。

R P Cは、これらのモデルの中で、手続き型言語との親和性が高く、かつ通信をプログラミング上ほとんど意識する必要がないために、様々な分散計算システムの構築に使われている。実用化されているR P Cの例としては、S u n R P C (S u n M i c r o s y s t e m s、I n c.著の文献「R P C : R e m o t e P r o c e d u r e C a l l P r o t o c o l s p e c i f i c a t i o n :

V e r s i o n 2」Network Working Group R F C—1 0 5 7、J a n . 1 9 8 8 (以下、引用文献1)の第3章に記載)や、C O R B A (T . J . M o w b r a y と W . A . R u h 著の文献「I n s i d e C O R B A」A d d i s o n — W e s l e y、1 9 9 7 (以下、引用文献2)の1.5節に記載)がある。

R P Cは、第1のオブジェクト(クライアントオブジェクト……以下単にクライアントと呼ぶ)で第1の手続きを呼び出すと、第2のオブジェクト(サーバオブジェクト……以下単にサーバと呼ぶ)内で第2の手続きを呼び出す仕組みによって実現される。サーバがクライアントに提供する第2の手続きを本願明細書では遠隔手続きと呼ぶ。なお、プログラミング上通信を意識させないために、第1の手続きと第2の手続きには同じ名前(第1の手続き名と呼ぶ)を用いるのが普通である。この際、第1の手続きに与えられた引数群は第2のオブジェクトに通信され、第2の手続きに与えられる。また、第2の手続き実行後の返り値は、第1のオブジェクトに通信され、第1の手続きの返り値となる。この仕組みを実現する手順の一例は、以下の通りである。プログラマが、手続きインターフェース定義言語(I n t e r f a c e D e f i n i t i o n L a n g u a g e ; I D L)を用いて、遠隔手続きの名前である第1の手続き名と、該遠隔手続きの型(引数群の型および返り値の型)を記述する(以下、I D L D e s c r i p t i o nと言う)。I D Lにより記述された遠隔手続きの名前及び型など、遠隔手続きのI D L記述をサーバのプログラマが書き、このI D L記述を入力として動作するI D Lコンパイラにより、I D L記述を第1のオブジェクトおよび第2のオブジェクトの原始プログラム(

ソースコード) のプログラミングに用いるプログラミング言語 (以下、ソースプログラミング言語) に翻訳する。この翻訳によって、多くの場合3つの出力が得られる。第1の出力は、IDL記述中の第1の手続きの名前および型をソースプログラミング言語で記述した「RPCヘッダファイル」である。RPCヘッダファイルには、第1のオブジェクトおよび第2のオブジェクトを構築する際に利用される型宣言が格納されている。なお、ソースプログラミング言語の種類によっては、RPCヘッダファイルを必要としない場合がある。

第2の出力は、「クライアントスタブ」と呼ばれるソースコードである。クライアントスタブには第1の手続きの定義 (すなわちコード列) が含まれ、第1のオブジェクトを構築する際の一部として用いられる。第3の出力は、「サーバスタブ」と呼ばれるソースコードである。サーバスタブには、クライアントスタブが発する通信を受けて第2の手続きを呼び出す手続きの定義が含まれ、第2のオブジェクトを構築する際の一部として用いられる。

第1のオブジェクトの構築は、第1のオブジェクト固有の機能を実現するためのソースコードとクライアントスタブとをコンパイルし、RPCランタイムライブラリと呼ぶスタブを補助するライブラリをリンクすることによって行われる。また、第2のオブジェクトの構築は、第2のオブジェクト固有の機能を実現するためのソースコードとサーバスタブとをコンパイルし、RPCランタイムライブラリをリンクすることによって行われる。そして、実行時には、以下に述べる手順でクライアントスタブとサーバスタブが通信することによって、第1の手続きの起動を第2の手続きの起動に対応づける。クライアントスタブとサーバスタブとの通信には、オペレーティングシステムおよびハードウェアが提供する、ネットワーク通信機能またはプロセス間通信機能が用いられることが多い。

クライアントスタブの第1の手続きの内容は、「第1の手続き名と、すべての引数をメモリ上の表現から通信データに変換し、サーバスタブに送信する。そして、サーバスタブから完了の通知があるのを待つ。完了の通知があったら、第1の手続きを終了する。サーバスタブから戻り値を受け取ったら、この戻り値を第1の手続きの戻り値とする。」という処理である。なお、サーバスタブが実行時にどの計算機上で動作し、どのように通信すべきか (通信プロトコル、通信ポートの選択等

）は、第1のオブジェクトの起動時引数等で指定され、第1の手続きに間接的に渡されることが多い。一方、サーバスタブの手続きの内容は、「第1の手続き名と、引数群を通信相手から受け取ったら、引数群を通信データからメモリ上の表現へ変換し、第1の手続き名に対応する第2の手続きを変換後の引数群を引数として呼び出す。第2の手続きが完了したら、通信相手に完了を通知する。この際第2の手続きに返り値があれば、この返り値をメモリ上の表現から通信データに変換し、通信相手に送信する。」という処理である。ここで通信相手とは第1のオブジェクトのクライアントスタブである。第1のオブジェクトが複数存在し、対応する第2のオブジェクトは1つ存在する、という構成もよく用いられるので、サーバスタブの手続きは、複数の通信相手を区別して動作することが多い。

なお、通常、1つのIDL記述には複数の遠隔手続きを記述できる。この場合、クライアントスタブにも複数の手続きが含まれる。また、サーバスタブには、該複数の手続きを区別し、該複数の手続きの呼び出しのそれぞれを、第2のオブジェクト中の複数の手続きの1つに対応づけ、呼び出す手続きの定義が含まれる。

次に、RPCの一般的な利用方法について説明する。

RPCは、クライアント・サーバ形態の分散処理によく用いられる。すなわち、サーバがクライアントに提供する機能の一つ一つを遠隔手続きとして定義しておき、クライアントが必要に応じて該遠隔手続きを呼び出すことによってサーバの機能を利用する。また、RPCはオブジェクト指向のプログラミングと親和性が良い（オブジェクト指向のRPCと呼ぶ）。オブジェクトのメソッド（オブジェクトを操作するインタフェース）一つ一つを遠隔手続きに対応させることにより、オブジェクト間の呼び出しを単一プロセス内・複数プロセス間・複数計算機間の区別なく行うことが可能となる（ここでプロセスとは、オペレーティングシステムが提供する実行単位である）。オブジェクト指向のRPCの代表例に、前述の引用文献2に記載のCORBAがある。

クライアント・サーバ形態のRPCでも、オブジェクト指向のRPCでも、プログラマが遠隔手続きとする手続きは、主として設計上の理由から、単一の機能を提供する手続きである傾向が強い。換言すると、遠隔手続きはそれぞれ独立した機能を提供しており、遠隔手続き群の組み合わせによって威力を発揮するように設計さ

れる。この設計方法により、設計後にクライアントの処理要求が変わっても、クライアント・サーバ間のインタフェースに変更をおよぼすことなくクライアントの処理変更や機能拡張が容易に可能となる。

この傾向は、オブジェクト指向のR P Cでも同じである。オブジェクト指向ではもともとメソッドとして、あるオブジェクトの操作に機能上必要十分な一組のインタフェースを用意することが多い。該オブジェクトの利用側のプログラムは、これらのメソッドを組み合わせることで該オブジェクトを操作する。この設計方法により、オブジェクトの利用側の柔軟な処理変更や機能拡張が容易に可能となる。すなわち、R P Cにおいては、遠隔手続きを多数組み合わせることで処理を進めるプログラム形態が一般的に用いられている。

【発明が解決しようとする課題】

遠隔手続きを多数組み合わせることで処理を進めるプログラムを作成すると、遠隔手続きに要する処理が大きいために、そのプログラムの目標とする性能が得られない場合がある。この最大の理由は、遠隔手続きのそれぞれが計算機間、あるいは単一計算機上のプロセス間の通信を伴うためである。遠隔手続きの呼び出しにかかる時間は、通常の手続き呼び出しにかかる時間に比べ、単一計算機内の通信で数10倍以上、複数の計算機間の通信では10、000倍程度（例えば、100MHzの計算機で手続き呼び出しに10サイクルかかるとすると、1回の手続き呼び出しは100ナノ秒。一方典型的なローカルエリアネットワークとTCP/IPプロトコルの組み合わせでは、一回の通信のレイテンシは少なくとも1ミリ秒程度）と非常に遅い。すなわち、手続き呼び出しと同じ頻度でR P Cを用いるプログラムを作成すると、最大10、000倍のオーダーで処理時間が大きくなる。

今後計算機の各部品がさらに高速化することが期待されているが、この場合、通常の手続き呼び出しとR P Cとの時間差はより大きくなると予想される。この理由は、プロセッサの速度向上が主に命令実行速度（毎秒あたりの実行命令数）の向上によるのに対し、ネットワークの速度向上が主にスループット（毎秒あたりの通信バイト数）の向上による。遠隔手続きによって起こる通信は、前述のように引数および返り値の通信であるため、一つ一つの通信データ（パケット）の長さは比較的小さい（多くは数バイトから数キロバイト）。

加えて、遠隔手続き毎の同期のために、通信が断続的にしか起こらない。このように小さなパケットが断続的に流れる状況では、高い通信スループットが得られにくい。

このため、プログラマが記述したクライアントのソースコードが多数のRPCを含んでいても、複数のRPCをまとめて実行し、実際に行う通信の回数を減らし、これにより通信のレイテンシを削減するとともに通信のスループットを向上させる最適化が求められる。この最適化を、プログラマ自身の手で行うのではなく、コンピュータ・プログラムが自動的に行えるようにすることが課題である。この課題が解決されることにより、上記RPCの一般的な利用方法と合致した効率のよいプログラム設計・開発を行いつつ、実行時の性能を向上させることが可能となる。

より具体的には、以下の(A)から(G)が解決すべき課題である。

(A) ストリーム通信やデータグラム通信におけるまとめあげと異なり、一連の遠隔手続きをまとめあげようとする場合には、クライアントがどのように該一連の遠隔手続きを利用しているかを知ることが不可欠である。この理由は、遠隔手続きがもともとの仕様上同期を伴うためと、遠隔手続きが一般に返り値を持つためである。

遠隔手続きは同期を伴うため、クライアントから依頼された遠隔手続きを単純に遅延させようとする、クライアントの処理が停止してしまい、処理が先に進まない。また、遠隔手続きの返り値は遠隔手続きを実行しなければ得られず、かつ該返り値はクライアントの遠隔手続き以降の処理に利用される可能性があるため、遠隔手続きを単純に遅延させるとクライアントの処理が正しく進まない。

(B) サーバが複数の遠隔手続きを提供しており、クライアントがこれらの遠隔手続きを適宜実行する状況が一般的であるが、このような状況で、複数のRPCを1つのRPCにまとめる場合、サーバ側に、該1つのRPCを処理する新たな遠隔手続きを追加して、サーバを拡張する必要がある。

また、このような状況で、単にあらゆる組み合わせを新たな遠隔手続きとしてサーバに追加しようとする、組み合わせの数が膨大になり実現できない。どの遠隔手続きをどのような順序で組み合わせ、1つの新たな遠隔手続きにするかを決定するためにクライアントのソースコードを解析する必要がある。

(C) サーバが複数の遠隔手続きを提供しており、クライアントがこれらの遠隔手続きを適宜実行する状況で、複数のRPCを1つのRPCにまとめる場合、該1つのRPCを処理する新たな遠隔手続きを呼び出せるようにするため、サーバのIDL記述に該新たな遠隔手続きの名前と型を追加する必要がある。

この際、1つのRPCの戻り値をクライアントのローカルな変数とあわせて次のRPCを起動することが一般的であると予想される。このため、該新たな遠隔手続きの名前と型を決定するためにクライアントのソースコードを解析し、どのようなクライアントのローカル変数が該新たな遠隔手続き内で使われるかを決定する必要がある。

(D) また、クライアントのソースコードを解析し、RPCを通常の手続きと区別するためには、IDL記述とクライアントのソースコードとを照合する必要がある。IDL記述は、システムによって提供の方法がまちまちであるため、各種の提供方法に対応する必要がある。すなわち、SunRPCやDCE RPC等の遠隔手続き呼び出しシステムの場合のように、IDL記述がサーバのソースコードと別に提供される場合に対処しなければならない。

(E) 例えば、CORBAでサーバに問い合わせることによりサーバが提供する遠隔手続きの名前と型がわかる場合のように、IDL記述がサーバのオブジェクトに（概念的に）埋め込まれている場合に対処しなければならない。

(F) 例えば、Java言語（JavaはSun Microsystems Inc.の登録商標である。）で遠隔手続きの名前と型がサーバのソースコード中に記述されている場合のように、IDL記述がサーバのソースコードに（概念的に）埋め込まれている場合に対処しなければならない。

(G) 複数の遠隔手続きをまとめあげる場合に、遠隔手続きの性質（手続きによる副作用の有無、手続きと手続きの並列実行可能性、手続きと手続きの実行順序交換可能性等）が分かると、まとめあげ方の選択肢が広がり、結果としてまとめあげの効率があがる。このため、クライアントが複数の遠隔手続きを連続的に発行する実行列を検出した際に、該実行列を実行する1つの新たな遠隔手続きとすることを助けるため、遠隔手続きの性質をIDL記述中に表現する方法を備えなければならない。

発明の要約

(SUMMARY OF THE INVENTION)

従って、本発明の目的は上記（Ａ）から（Ｇ）の課題を解決し、上記ＲＰＣの一般的な利用方法、および今後の技術進歩の傾向に鑑みて、遠隔手続きが多数呼び出される場合のプログラム実行性能を向上する方法を提供し、この方法を用いたプログラムの作成と実行を容易にすることである。

ＲＰＣが通常の手続き呼び出しに比べて時間がかかる理由は、前述のように、遠隔手続きのそれぞれが通信を伴うためである。遠隔手続きの一連の呼び出し群をひとつにまとめて通信を行い、該一連の呼び出しを一括してサーバ側で実行する。これにより遠隔手続き一個当たりの通信のレイテンシが減少する。さらに、引数および戻り値を入れるパケットがまとめあげにより大きくなるため、従来よりも高スループットで引数および戻り値の通信が可能となる。

特に、上記（Ａ）から（Ｇ）の課題を解決するため、本発明は以下の（ａ）から（ｇ）の各手段を用いる。

（ａ）サーバのみ、またはクライアントのみを改変することによりＲＰＣの最適化を行うのではなく、サーバとクライアントの両方を改変することにより、複数のＲＰＣをできるだけ少ない回数（例えば、１回）のＲＰＣにまとめる。

（ｂ）クライアントのソースコードを解析してクライアントが複数の遠隔手続きを連続的に発行する実行列を検出し、該実行列を実行する１つの新たな遠隔手続きをサーバに追加する。

（ｃ）クライアントのソースコードを解析してクライアントが複数の遠隔手続きを連続的に発行する実行列を検出し、該実行列を１回のＲＰＣで実行する新たな遠隔手続きのインターフェースに相当する引数および戻り値を決定し、該新たな遠隔手続きの引数および戻り値をＩＤＬ記述に追加する。

（ｄ）主にＩＤＬ記述がサーバのソースコードと別に提供される場合のため、クライアントのソースコードとＩＤＬ記述を入力として、改変したクライアントのソースコードと、改変したＩＤＬ記述と、追加すべきサーバのソースコードを出力する。

(e) 主にIDL記述がサーバのオブジェクトに（概念的に）埋め込まれている場合のため、クライアントのソースコードとサーバのオブジェクトを入力として、改変したクライアントのソースコードと、サーバに追加すべきソースコードを出力する。

(f) 主にIDL記述がサーバのソースコードに（概念的に）埋め込まれている場合、上記最適化をプログラムの新たな手間を生じさせずに行うため、クライアントのソースコードとサーバのオブジェクトを入力として、改変したクライアントのソースコードと、サーバに追加すべきソースコードを出力する。

(g) クライアントが複数の遠隔手続きを連続的に発行する実行列を検出した際に、該実行列を実行する1つの新たな遠隔手続きとすることを助けるため、遠隔手続きの性質をIDL記述中に表現する。

より詳細には、本発明は、IDL記述とともにクライアントのソースコードを入力として、RPCを最適化するRPCオブティマイザを用いる。RPCオブティマイザは、IDL記述とクライアントのソースコードを入力として、変更したIDL記述と変更したクライアントのソースコード、およびサーバの追加のソースコードを出力するコンパイラである。

RPCオブティマイザはクライアントのソースコードを解析して、一連の遠隔手続き呼び出しのうちまとめあげによる性能向上が期待でき、かつまとめあげが可能なRPC列を抽出し、該RPC列を新たな遠隔手続きとして、そのインタフェースをIDL記述に加える。

一方、クライアントのソースコードの該RPC列は、該新たな遠隔手続きを利用するコードに変更する。そしてサーバの追加のソースコードとして、該新たな遠隔手続きの本体（既存のRPC列と、クライアントのソースコードから抽出した処理とからなる）を生成する。

一般に、遠隔手続きは通常の手続きと区別なく利用されるため、クライアントのソースコード中で遠隔手続きばかりが連続して並ぶ（またはループによって繰り返し呼ばれる）ことはむしろ稀である。逆に、クライアント内の変数の参照や変更、返り値による分岐、クライアント内の手続きの呼び出しと、ある程度混じりあって呼び出されることが多いと予想される。この場合を扱うために、RPCオブティ

マイザはクライアントのソースコード中のデータの流れを解析すること（データ依存性解析）によってまとめあげ可能なR P C列を抽出する。この際のデータ依存性解析には、従来からコンパイラで用いられているデータフロー解析と、データ型に着目したデータ依存性解析を用いる。このデータ依存性解析により、遠隔手続きとクライアント内処理との切り分けが困難な場合には、R P Cオブティマイザはクライアント内処理の一部を含めて新たな遠隔手続きとすることで、まとめあげを行う。

さらに、引数や返り値の大きさが実行時に決まる遠隔手続きもある。この種の遠隔手続きのまとめあげの効果を知り、まとめあげをおこなうかどうかを決定するには、実行時の情報を得る必要がある。このため、本発明では、実行中の各遠隔手続きの起動引数および返り値の大きさの統計を記録する。この記録は、R P Cランタイムライブラリまたはスタブで行うことができる。また、オペレーティングシステム（O S）等、より下位のレイヤで統計を記録することも可能である。該記録は、R P Cオブティマイザが次のコンパイル時に利用する。

以上の手段により、一連のR P Cをまとめあげ、これによりR P Cにおいて遠隔手続きが多数呼び出される場合の性能を向上し、以て遠隔手続きを用いたプログラムの作成を容易にする。

図面の簡単な説明

(BRIEF DESCRIPTION OF THE DRAWINGS)

【図1】

本実施例の構成の概略を示すブロック構成図。

【図2】

本発明を適用する分散コンピュータ・システムの全体構成図。

【図3】

従来の遠隔手続き呼び出しのコンパイルおよび実行内容を示すブロック構成図。

【図4】

コンパイル済みサーバを用いた従来の遠隔手続き呼び出しのコンパイルおよび

実行内容を示すブロック構成図。

【図 5】

R P C オプティマイザのブロック構成図。

【図 6】

R P C オプティマイザのデータ構造を示す構成図。

【図 7】

I D L ソースコードとクライアントソースコードの一例を示す図。

【図 8】

R P C ヘッドファイルとクライアントスタブの一例を示す図。

【図 9】

サーバスタブの一例を示す図。

【図 1 0】

新 I D L ソースコード、新クライアントソースコード、追加サーバソースコードの一例を示す図。

【図 1 1】

ループによって繰り返し実行される R P C を検出する方法（第 1 の方法）の処理手順を示すフローチャート。

【図 1 2】

基本命令列の中に登場する R P C の数が 2 以上である部分を検出する方法（第 2 の方法）の処理手順を示すフローチャート。

【図 1 3】

拡張可能ディスパッチャのブロック図。

【図 1 4】

拡張後の拡張可能ディスパッチャのブロック図。

【図 1 5】

インタプリタつきディスパッチャのブロック図。

【図 1 6】

階層構造サーバのブロック図。

【図 1 7】

通信計測部を含むR P Cオプティマイザのブロック図。

【図18】

拡張IDLの構文の一例を示す図。

【図19】

第1の変形例のブロック構成図。

【図20】

第2の変形例のブロック構成図。

【図21】

第3の変形例のブロック構成図。

発明の実施の形態

(DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS)

以下、本発明の実施の一形態を、図面を参照しながら説明する。

全体構成

図1と図2を用いて、本実施例の概略を説明する。

まず、図2の本実施例の全体構成201は、本実施例が好適に用いられる分散コンピュータ・システムであり、ネットワーク202と、ネットワーク202によって相互接続された複数の計算機203、203'、203''、…からなる。

ネットワーク202は、ある団体（企業や学校や類似の団体）の全体や位置部門でよく使用されるLANでもよく、また地理的に分散した複数の地点を結合するWANの一部または全部でもよい。またネットワーク202は、計算機間結合網や並列計算機内部のプロセッサ要素間の結合網でもよい。

計算機203、203'、203''、…は、いわゆるパーソナル・コンピュータ、ワークステーション、並列計算機、大型計算機等、任意のコンピュータでよい。また、クライアント204、204'、204''、…を動作させる計算機203、203'、203''、…は、サーバ205、205'、…と通信する機能を持つものであれば種別を問わない。すなわち各種コンピュータ端末装置、携帯通信端末（いわゆるパーソナル・デジタル・アシスタンスPDAやハンドヘルド・パーソナル・コンピュータHPC）、ネットワーク・コンピュータなどでも差し支え

ない。

サーバ205、205'、…およびクライアント204、204'、204''、…はいずれも、計算機203、203'、203''、…で実行される実行中のプログラムまたはプログラム部品（オブジェクト）である。サーバ205、205'、…は遠隔手続きと呼ばれる側のオブジェクト、クライアント204、204'、204''、…は遠隔手続きを呼ぶオブジェクトである。サーバとクライアントの区別は、ある遠隔手続きに着目した場合の関係であり、場合によってはあるサーバが別のサーバのクライアントになったり、2つのオブジェクトがお互いにサーバとクライアントである（ある遠隔手続きについては第1のオブジェクトがサーバで第2のオブジェクトがクライアント、また別の遠隔手続きについては第2のオブジェクトがサーバで第1のオブジェクトがクライアント）であって差し支えない。

計算機203、203'、203''、…は、それぞれ1個以上のクライアントまたは1個以上のサーバまたはその両方を動作させる。本実施例の全体構成201では、少なくとも1つのサーバと少なくとも1つのクライアントが存在する。なお、図2に示した計算機203、203'、203''、…、クライアント204、204'、204''、…、およびサーバ205、205'、…の数と構成は、一例として示したものである。また、本発明の実施は、クライアントまたはサーバのオペレーティング・システム、サーバ間またはサーバ・クライアント間のネットワークの種類およびネットワークの物理層プロトコルやトランスポート層プロトコル、もしくはサーバとクライアントが単一のコンピュータ上にあるか異なるコンピュータ上にあるかには依存しない。また、本実施例の構成の一方法として、ネットワークに繋がっていない1台の計算機のみがあつて、サーバとクライアントが異なるプロセス上で動作しても差し支えない。この場合には本発明は、2つのプロセスの間の通信を最適化する方法として利用できる。

図1は、クライアントのコンパイル時から、クライアントとサーバの実行時に至る、本実施例の利用の流れを示している。なお、本実施例では説明を分かりやすくするため、コンパイラ・リンカ、IDLコンパイラなど公知のプログラムと組み合わせる例を示すが、本発明はこれに限定されるものではない。例えば、IDLコンパイラと本実施例のRPCオプティマイザの両機能を持つ1つの実行プロ

グラム、コンパイラと本実施例のRPCオブティマイザの両機能を持つ1つの実行プログラム等は、本発明の他の実施形態として容易に考える。

また、IDL記述は、IDL記述ファイルに格納されて提供される場合の他、サーバオブジェクト内に格納されている場合、サーバオブジェクトとの通信によって得られる場合、IDL記述を格納するサーバ（インタフェースリポジトリ）との通信によって得られる場合等の種類がありうる。本実施例では説明を簡単にするため、IDL記述がIDL記述ファイルであるIDLソースコード103に格納されて提供される場合について説明するが、上に示したような他の方法でIDL記述が提供されても差し支えない。

RPCオブティマイザ101は、クライアントを構成するために用いられるクライアントソースコード102、102'、…と、サーバの提供する遠隔手続き群のインタフェースをIDLで記述したIDLソースコード103を入力とし（150—152）、RPCを最適化し、その結果、新クライアントソースコード105、105'、…、新IDLソースコード106、および追加サーバソースコード107（153、154、155、156）を出力する。

IDLコンパイラ108は、新IDLソースコード106を入力（157）として動作し、「従来の技術」で説明したクライアントスタブ109、RPCヘッダファイル110、およびサーバスタブ111を出力する（158、159、160）。

新クライアントソースコード105、105'、…とクライアントスタブ109とRPCヘッダファイル110は、コンパイラ・リンカ112によってコンパイル及びリンクされ（161、162、163、164）、クライアントの実行可能プログラムであるクライアントオブジェクト114となり出力される（165）。また、追加サーバソースコード107とサーバスタブ111とRPCヘッダファイル110は、コンパイラ・リンカ113によってコンパイル及びリンクされ（168、167、166）、追加サーバオブジェクト115となり出力される（169）。以上がコンパイル時の動作である。

図1に示す実行時の一例では、クライアントオブジェクト114が計算機203'上で実行される（170）。一方、追加サーバオブジェクト115はサーバオ

プロジェクト104と動的リンカ116によって動的にリンクされ(171、172) 計算機203上で実行される(173)。実行時に起こるRPC(174)は、IDLソースコード103にもともと宣言されていた遠隔手続きと、新IDLソースコード106で宣言された遠隔手続きの少なくとも一方が起こりうる。

本発明の理解を容易にするため、本実施例とは異なる方法でRPCを行う従来のクライアントとサーバのコンパイルおよび実行の流れを、図3および図4に示す。

IDLコンパイラ304は、IDLソースコード302を入力とし(350)、クライアントスタブ305、RPCヘッダファイル306、およびサーバスタブ307を出力する(351、352、353)。

クライアントソースコード301、301'、…とクライアントスタブ305とRPCヘッダファイル306は、コンパイラ・リンカ308によってコンパイル及びリンクされ(354、355、356、357)、クライアントの実行可能プログラムであるクライアントオブジェクト310となり出力される(362)。また、サーバソースコード303、303'、…とサーバスタブ307とRPCヘッダファイル306は、コンパイラ・リンカ309によってコンパイル及びリンクされ(358、359、360、361)、サーバオブジェクト311となり出力される(363)。以上がコンパイル時の動作である。

図3に示す実行時の一例では、クライアントオブジェクト310が計算機203'上で実行される(364)。一方サーバオブジェクト311は計算機203上で実行される(365)。実行時に起こるRPC(366)は、IDLソースコード302にもともと宣言されていた遠隔手続きである。

CORBAなどの分散オブジェクト技術では、サーバはパッケージとしてコンパイル済みのプログラムが提供され、クライアントをコンパイルしてサーバと通信させる、というコンパイル・実行形態もある。このコンパイル・実行形態を図4に示す。

IDLコンパイラ404は、IDLソースコード402を入力として動作(450)し、クライアントスタブ405(451)とRPCヘッダファイル406(452)を出力する。

クライアントソースコード401、401'、…とクライアントスタブ405とRPCヘッダファイル406は、コンパイラ・リンカ407によってコンパイル及びリンクされ(453、454、455、456)、クライアントの実行可能プログラムであるクライアントオブジェクト408となり出力される(457)。以上がコンパイル時の動作である。

図4に示す実行時の一例では、ユーザがクライアントオブジェクト408を計算機203'上で実行し(458)、サーバオブジェクト403を計算機203上で実行する(459)。実行時に起こるRPC(460)は、IDLソースコード402にもともと宣言されていた遠隔手続きである。

内部構造

次に、図5を用いて、本発明によるRPCオブティマイザ101の内部構造を説明する。RPCオブティマイザ101は、レキシカルアナライザ503、パーザ504、内部表現生成部505、IDLレキシカルアナライザ506、IDLパーザ507、RPC表生成部508、中間コード変換部509、ソースコード生成部510、IDLソースコード生成部511の各部からなる。

レキシカルアナライザ503は、クライアントソースコード501、501'、…を入力として受け取り(550、550'、…)、字句解析を行う。クライアントソースコード501、501'、…の文字を逐一解析し、予約語、名前(変数名、手続き名等)、区切り記号、定数等の語句の列に分解する。この字句解析に関しては、すでによく知られており、例えば文献「A.Aho、J.Ullman”Principles of Compiler Desing、”Addison—Wesley Publishing Company、April 1977(以下、引用文献3)のpp.10」に記載されている。なおこの段階で、あとで新クライアントソースコード512、512'、…を生成する際の便宜のため、どの語句の列がどのクライアントソースコードから来たかを記録してもよい。

パーザ504は、レキシカルアナライザ503の出力である語句の列を受け取り、クライアントソースコードの言語の文法に従って、語句の列をパーズツリーと呼ばれるデータ構造に構成し、パーズ結果520に格納する(551)。パーズツリーとは、文法上の構成要素である式、文、ブロック、手続き、プログラム等を表

現するためのデータ構造である。多くのプログラミング言語が再帰的な構文を許す文脈自由文法である（または文脈自由文法に近い）ため、パーズツリーは木構造のような再帰的な構造を表現できるデータ構造で作られることが多い。パーズ処理、またはパーズングに関してもすでに良く知られており、例えば引用文献3（pp. 12）に記載されている。パーズ結果520の構造は、ソースコードの言語の構造（型宣言、手続き定義、文、式、変数、定数等）にそってクライアントソースコード501、501'、…を表現した木構造である。

内部表現生成部505は、パーザ504の出力であるパーズ結果520を入力として（552）、最適化処理に都合のよいデータ構造に分解、再構成し、格納する。この処理も、中間言語生成として良く知られている。例えば引用文献3（pp. 13）に記載されている。内部表現生成部505は、命令表521、基本ブロック表522、複合ブロック表523、環境表524の4種類のデータ構造を出力する（553）。これらの内部構造については、あとで図6を用いて詳細に説明するが、概略は以下の通りである。

命令表521はパーズツリーのうち、実行文をいくつかの基本命令（参照、加減算等の演算、代入、手続き呼び出し、分岐、繰り返しなど）の列で表現したデータ構造である。基本ブロック表522は、命令表521に対して分岐や繰り返しなどの制御構造に着目した、基本ブロックと呼ぶ一連の基本命令列に区分した結果を保持する。基本ブロックの概念は例えば引用文献3（pp. 412）に記載されている。複合ブロック表523は、いくつかの関連する基本ブロックの集まりを表現するデータ構造であり、一個の手続きや、ソースコード上のブロック（例えば、C言語の開き中括弧から対応する閉じ括弧まで、Pascal言語のbeginからendまでの間の文）を格納する。環境表524は、基本ブロックや複合ブロックの中で使われる変数に関する情報を格納する。

IDLレキシカルアナライザ506は、IDLソースコード502を入力として受け取り（554）、IDLソースコードの字句解析を行う。すなわちIDLソースコード502の文字を逐一解析し、予約語、名前（変数名、手続き名等）、区切り記号、定数等の語句の列に分解する。

IDLパーザ507は、IDLレキシカルアナライザ506の出力である語句

の列を受け取り、IDLの文法に従って、語句の列をパーズツリーに呼ばれるデータ構造に構成する。IDLはほとんどが宣言であるため、このパーズツリーの内容は型宣言と手続きの宣言が主要な部分を占める。

RPC表生成部508は、IDLパーザ507の出力を、RPCの手続き名、入力引数(RPCの呼び出し時にクライアントからサーバに渡されるデータ)、出力引数(RPCの終了時にサーバからクライアントに返されるデータ、RPCの返り値も含む)等からなるRPCのインタフェース、およびユーザ定義の型宣言に再構成し、結果をRPC表527へ格納する(556)。IDLパーザ507の出力に型宣言があれば、これもRPC表527へ格納する。RPC表527の内部構造についてもあとで図6を用いて詳細に説明するが、IDLソースコード502に列挙されたRPCインタフェースの型、およびユーザ定義の型宣言を列挙したデータ構造である。

中間コード変換部509は、パーズ結果520、命令表521、基本ブロック表522、複合ブロック表523、環境表524、RPC表527を用いて、RPCの最適化を行う。詳細な処理はあとで図11と図12を用いて行うが、概略は以下の処理である。RPC表527を照らし合わせながら命令表521を検査する(558、564)ことにより、命令表521の基本命令列うち、RPCが多発する第1の基本命令列を検出する。この第1の基本命令列を、新たな第1の手続きとしてまとめあげる。RPCが多発する基本命令列の発見には、本実施例では2つの方法を用いる。第1の方法はループによって繰り返し実行されるRPCを検出する方法、そして第2の方法は、基本ブロック内の一部または全部の基本命令列の中に登場するRPCの数が2以上である部分を検出する方法である。なお、これら2つの方法は、代表的な例として示したものであり、本発明の範囲を限定するものではない。上記2つの方法の他、一連の基本命令列中のRPCとRPC以外の割合を計算し、一定以上RPCの割合が多い部分を検出する方法や、ローカルなコード最適化と組み合わせてRPCを移動させ、もともと連続していなかったRPCを近くに寄せる方法など、いくつかの方法が考えられる。これらの一部または全部を組み合わせ使用しても差し支えない。

次に、第1の命令列を命令表521から取り除き、かわりに第1の手続きへの

手続き呼び出しを行う基本命令列を挿入し（５５８）、パーズ結果５２０も対応させて変更する（５５７）。これに伴って基本ブロック、複合ブロック、変数群の構成に変更があった場合には、基本ブロック表５２２、複合ブロック表５２３、環境表５２４も変更する（５５９、５６０、５６１）。

第１の手続きを構成する基本命令列は、手続きとして必要な最初の部分（主に入力引数の操作）と最後の部分（主に出力引数の操作と制御の移動）を追加した後、命令表５２１と同じ構造を持つサーバ側命令表５２５に格納する（５６２）。また第１の基本命令列に対応するパーズ結果５２０の部分サーバ側パーズ結果５２６に移動し、同じく手続きとして必要な最初と最後の部分を追加する（５６３）。さらに第１の手続きのインタフェースをRPC表５２７に格納する（５６４）。

第１の手続きの入力引数は、第１の基本命令列が使用する、第１の基本命令列の外で定義された変数群とする。また第１の手続きの出力引数は、第１の基本命令列で定義または変更され、第１の基本命令列以外の基本命令が参照する変数群とする。この結果、第１の手続き中で参照、変更される変数は第１の手続きへのまとめあげ以前と同じ動作となる。RPCが多発する基本命令列を検出し、一旦まとめあげを試みても、入力引数および出力引数のデータ量が大量になってしまい、効率が損なわれる恐れがでる場合もある。この場合には、そのまとめあげは中止する。

以上のまとめあげの処理を繰り返し行い、さらなるまとめあげができなくなった時点で中間コード変換部５０９を終了する。

ソースコード生成部５１０は、中間コード変換部５０９が変更したパーズ結果５２０、サーバ側パーズ結果５２６、命令表５２１、サーバ側命令表５２５、基本ブロック表５２２、複合ブロック表５２３、および環境表５２４を用いて、新クライアントソースコード５１２、５１２'、…と追加サーバソースコード５１３を出力する（５６５）。これは、パーズツリーからソースコードプリティプリンティングとよばれる処理と共通の処理である。具体的には、パーズ結果５２０またはサーバ側パーズ結果５２６を深さ優先で探索し、木構造のノードを通過する順にソースコードに変換し、出力する。一つ一つのノードは、ソースコードの言語の基本要素（単項演算や二項演算、変数の参照、if文、for文、代入文、手続き呼び出し

、ブロック、手続き、プログラム等) であるので、各ノードのソースコードへの変換は機械的に行うことができる。主としてパース結果 5 2 0 から新クライアントソースコード 5 1 2、5 1 2'、…が生成され (5 6 7、5 6 7')、サーバ側パース結果 5 2 6 から追加サーバソースコード 5 1 3 が生成される (5 6 8)。

IDLソースコード生成部 5 1 1 は、RPC表 5 2 7 から新IDLソースコード 5 1 4 を生成する (5 6 6、5 6 9)。RPC表 5 2 7 はIDLソースコード 5 0 2 を翻訳して保持しているデータ構造であるため、IDLソースコード生成部 5 1 1 はRPC表生成部 5 0 8 の逆の操作を行うことでRPC表 5 2 7 の内容からIDLを再構成できる。なお、RPC表 5 2 7 は中間コード変換部 5 0 9 による最適化のための変更を経ているので、出力される新IDLソースコード 5 1 4 には、新IDLソースコード 1 0 6 で宣言されていたRPCインタフェースの他、中間コード変換部 5 0 9 が最適化の結果追加したRPCインタフェースが存在することになる。

データ構造

次に、図 6 を用いて、命令表 5 2 1、基本ブロック表 5 2 2、複合ブロック表 5 2 3、環境表 5 2 4、RPC表 5 2 7、および変数表 6 6 0 のデータ構造を説明する。

命令表 6 0 0 は命令表 5 2 1 の構造を表わしている。命令表 6 0 0 は 1 つ以上の命令表要素 6 0 1 からなり、命令表要素 6 0 1 が 1 つの基本命令を表わす。命令表要素 6 0 1 はさらに命令 ID 6 0 2、ターゲット 6 0 3、命令 6 0 4、オペランド A 6 0 5、オペランド B 6 0 6 の 5 個のフィールドからなる。命令 ID 6 0 2 は命令表要素 6 0 1 に与えられた番号である。ターゲット 6 0 3、オペランド A 6 0 5、オペランド B 6 0 6 は変数やデータ格納領域の名前である。命令 6 0 4 は、基本命令の種類で、「単項演算」(マイナス、論理 NOT など)、「二項演算」(加減乗除、二項論理演算、構造体参照等)、「条件付き分岐」、「無条件分岐」、「手続き呼び出しの引数指定」、「手続き呼び出し」、「代入」などを表わす。オペランド A 6 0 5、ターゲット 6 0 3 は操作対象、オペランド B 6 0 6 は命令 6 0 4 の引数であり、これらの解釈は基本命令の種類によって意味が異なる。例えば” IF A RELOP B GOTO L” という基本命令では、命令 6 0 4 に” IF

RELOP GOTO”を、オペランドA605とオペランドB606にそれぞれAとBを、ターゲット603にLを格納する。

基本ブロック表610は、基本ブロック表522の構造を表わしている。基本ブロック表610は、1つ以上の基本ブロック表要素611からなり、基本ブロック表要素611はさらに基本ブロックID612、開始命令ID613、終端命令ID614、次基本ブロック615、前基本ブロック616、環境ID617、DGEN変数表618、DKILL変数表619、DIN変数表620、DOUT変数表621、LIN変数表622、LOUT変数表623、LUSE変数表624、およびLDEF変数表625の14個のフィールドからなる。

ある基本ブロックについて、基本ブロックID612は、該基本ブロックを識別する番号である。開始命令ID613は、該基本ブロックに対応する基本命令列のうち、最初の基本命令の命令ID602である。終端命令ID614は、該基本ブロックに対応する基本命令列のうち、最後の基本命令の命令ID602である。次基本ブロック615は、次の基本ブロックの基本ブロックIDである。前基本ブロック616は、ひとつ前の基本ブロックの基本ブロックIDである。次基本ブロック615と前基本ブロック616は、2つ以上の基本ブロックIDを格納してもよい。環境ID617は、該基本ブロックに対応する変数群を格納する環境表524を指す。

DGEN変数表618、DKILL変数表619、DIN変数表620、DOUT変数表621、LIN変数表622、LOUT変数表623、LUSE変数表624、LDEF変数表625はいずれも、後述する変数表660の構造を持つ。

DGEN変数表618は、該基本ブロックで値が新たに定義される変数群の定義場所を格納する変数表660である。DKILL変数表619は、該基本ブロックで定義が失われる変数群について、それらの定義場所を格納する変数表660である。DIN変数表620は、該基本ブロック以前に定義された変数群の定義場所を格納する変数表660である。DOUT変数表621は、該基本ブロックで定義され、次の基本ブロックから参照される可能性がある変数群の定義場所を格納する変数表660である。DGEN変数表618、DKILL変数表619、DIN変数表620、DOUT変数表621の計算方法についてはすでに知られている。例

例えば引用文献3（pp.431—433）に記載されている。

LIN変数表622、該基本ブロックまたは該基本ブロック以降の基本ブロックで参照される変数群の定義場所を格納する変数表660である。LOUT変数表623は、該基本ブロック以降の基本ブロックで参照される変数群を格納する変数表660である。LUSE変数表624は、該基本ブロックで参照される変数群を格納する変数表660である。LDEF変数表625は、該基本ブロックで定義され、次のブロック以降で参照される変数群を格納する変数表660である。LIN変数表622、LOUT変数表623、LUSE変数表624、LDEF変数表625の計算方法についてはすでに知られている。例えば引用文献3（pp.489—490）に記載されている。

複合ブロック表630は、いくつかの関連する基本ブロックの集まりを表現する複合ブロック表523の構造を示している。複合ブロック表630は1つ以上の複合ブロック表要素631の配列である。ある複合ブロックについて、複合ブロックID632は該複合ブロックを識別する番号である。開始基本ブロックID633は該複合ブロックの始まりの基本ブロックの基本ブロックID612を格納する。終端基本ブロックID634は該複合ブロックの終わりの基本ブロックの基本ブロックID612を格納する。環境ID635は該複合ブロックに付随する変数群を格納する環境表524を指す。

環境表640は、基本ブロックや複合ブロックの中で使われる変数に関する情報を格納する環境表524の構造を示している。環境表は複数存在することがあり、環境ID641はそれぞれの環境表を識別する番号である。多くのプログラミング言語で、変数の有効範囲（スコープ）は階層構造となるので、親環境ID642でこの階層構造を表現する。属性643は該環境表の種々の負荷情報を保持する。環境内変数表644は環境表の持つ1つ以上の変数に関する情報を格納する。環境内変数表644の構造は後述の変数表660である。

PRC表650は、RPCの手続き名、入力引数、出力引数等からなるRPCのインタフェース、およびユーザ定義の型宣言を格納するRPC表527の構造を示している。PRC表650は0個以上のPRC表要素651と0個以上の型宣言要素658からなる。PRC表要素651はひとつのRPCインタフェースに対

応し、それぞれPRC名652、IN引数表653、OUT引数表654、属性655からなる。PRC名652はRPCの名前、IN引数表653は該RPCインタフェースの入力引数、OUT引数表654は該RPCインタフェースの出力引数であり、どちらも後述の変数表660の構造を持つ。属性655は、該RPCインタフェースに関する追加情報を保持する。追加情報の例としては、例外に関する情報、環境変数に関する情報でもよいし、IDLを後述のように拡張する場合には、RPCの最適化に利用可能な情報でもよい。型宣言要素658は、1つのユーザ定義の型を表わし、型名656と型情報657からなる。型名656は型の名前、型情報657は他の型宣言要素や基本型を用いて型名656の型の構造を表現した情報である。

変数表660は、変数群の名前及び負荷情報を保持する配列である。変数表660は1つ以上の変数表要素661からなり、変数表要素661はさらに変数名662、型663、属性664からなる。

なお、ここで説明しなかったパース結果520とサーバ側パース結果526、すなわちパースツリーの実現方法については引用文献3等多くの文献に示されており、パースツリーの生成を半自動化するツールもよく用いられるため、ここでは特に詳しく述べない。

最適化例

次に、図7、図8、図9、および図10を用いて、RPCオブティマイザ101が行うRPCの最適化の例を説明する。

`intf.idl` 700は、IDLソースコードの例である。行701—704がソースコード本体である。行701は、RPCのインタフェース群をひとまとめにして、`MyServer`という名前をつけて宣言している。このIDLソースコードを用いて定義されるサーバオブジェクトは、`MyServer`という型を持つ。行702は`MyServer`に含まれる第1の遠隔手続き`func1`のインタフェース定義である。`func1`の引数および返り値の型がこの行で定義される。すなわち、`func1`は、入力引数（予約語`in`で指定される）を1つとり、その型は`int`型（整数型）で名前は`i`である。また、`func1`の返り値も、`int`型である。同様に、行703には、`MyServer`に含まれる第2の遠隔手続

き `func2` のインタフェース定義がある。`func2` は引数 `key` と、引数 `value` をとる。引数 `key` は `inout`、すなわち入力引数であり出力引数でもある。引数 `value` は入力引数である。

`client.c` 750 は、`intf.idl` 700 で定義されるサーバを利用するクライアントのソースコードの例である。記述言語は C++ 言語を用いている。行 751 で、後述する IDL ヘッドファイル `intf.h` 800 を読み込む。`intf.h` 800 に `MyServer` 型が定義されている。行 752—762 が該クライアントの始まりである `main` 関数の定義本体である。この定義では、説明を分かりやすくする目的で、いくつかのライブラリ関数を用いて例を示す。行 754 は、該クライアントが、名前サービスまたは辞書から探し出すライブラリ関数を利用して、利用するサーバを探す処理である。”`MyServer`” という名前を指定して `lookupDirectory` ライブラリ関数を呼び出すことで、`MyServer` 型のサーバを得ている。行 755 はローカル変数 `count` の定義および 0 への初期化である。行 756—758 は `for` 文による繰り返し処理で、`MyServer` 型のサーバ `server` の遠隔手続きである `func1` を呼び出す。ここでは、入力引数 `i` の値を 0 から 100 まで変化させつつ、100 回の RPC を行う。同時に、`count` に `func1` の戻り値の合計を計算する。行 759 は `count` の値を出力するライブラリ `printf` 関数である。行 760 は、`server` の別の遠隔手続きである `func2` を呼び出す。続いて行 761 では、`server` の `func1` を呼び出す。以上が該クライアントの処理である。該クライアントでは、合計 102 回の RPC が実行される。

図 3 に示した従来の遠隔手続き呼び出しのコンパイルを行うと、IDL コンパイラ 304 によって `intf.idl` 700 から `intf.h` 800、`clientstub.c` 850、および `serverstub.c` 900 が出力される。`intf.h` 800、`clientstub.c` 850、`serverstub.c` 900 はそれぞれ、RPC ヘッドファイル 306、クライアントスタブ 305、サーバスタブ 307 に対応する。本例では、`intf.h` 800、`clientstub.c` 850、`serverstub.c` 900 とも C++ 言語である。なお、ここで示した変換は一例であり、出力である `intf.h` 8

00、`clientstub.c` 850、および `serverstub.c` 900はシステムによって異なって差し支えない。

`intf.h` 800は、`MyServer`型オブジェクトの型宣言を格納したファイルである。行801はRPCを行うオブジェクトの親クラスとして利用する`Object`型の宣言が入ったファイルである`Object.h`を読み込む。本例では、`Object`型はサーバとクライアントが通信するメソッド`call`を提供する（メソッド`call`については後で説明する）。行802で、`MyServer`型のオブジェクトが`Object`型の子クラスであることを宣言する。行803は、`intf.idl` 700の行702で定義された`func1`を、C++言語に翻訳したメソッド`func1`を宣言する。引数`i`は`int`型、返回值も`int`型である。行804は、`intf.idl` 700の行703で宣言された`func2`を、C++言語に翻訳したメソッド`func2`を宣言する。

引数`key`は入力引数でありかつ出力引数であるので、C++の参照型（&記号で指定される）を用いて宣言される。また、引数`value`は、`intf.idl` 700では`String`型として宣言されているが、C++言語では対応する型がないので、`char*`型（文字型へのポインタ型）として宣言される。行805は`MyServer`型の宣言の終わりである。

`clientstub.c` 850は、`intf.h` 800で宣言された`MyServer`型の2つのメソッド`func1`と`func2`の本体の定義を格納したファイルである。`clientstub.c` 850が格納する定義は、クライアントが用いるもので、RPCの呼び出し側のコードである。すなわち、`clientstub.c` 850の`func1`および`func2`は、クライアント内の第1の関数から呼び出されると入力引数を通信パケットに詰めてサーバに送って、サーバの返事を待ち、サーバからの返事の通信パケットから出力引数を取りだし、第1の関数に返す、という処理を行う。詳細は以下の通りである。行851は、`MyServer`型の宣言を得るため、`intf.h` 800を読み込む。行852から行861は`func1`の定義、行862から行870は`func2`の定義である。行852は、`MyServer`型のメソッド`func1`が、`int`型の入力引数を取り、`int`型の返回值を返す関数であることを宣言する。

行854は、サーバと通信するための通信パケットbufを新たにメモリ割り当てする。bufはBuffer型であり、通信パケットにさまざまな型の値を詰める操作と、さまざまな型の値を通信パケットから取り出す操作を提供する。行855はローカル変数rvalの宣言である。行856で、bufに、int型の値を格納するメソッドpackintを用いてiをbufに格納する。行857で、MyServerの親クラスであるObject型が提供するcallメソッドを用いて、サーバと通信を行う。この際、サーバ側で起動すべき遠隔手続きの名前”func1”と、入力引数が入ったbufをメソッドcallに与える。メソッドcallは、遠隔手続きの名前”func1”と bufをサーバに送信し、サーバからの応答を待つ。サーバからの応答は、再びbufに格納される。行858は、サーバからの返答からint型の戻り値を取り出し、ローカル変数rvalに代入する。この際、Buffer型の提供するunpackintを用いる。行859は不要になった通信パケットbufのメモリを開放する。行860で、rvalの値をfunc1の戻り値とし、制御を呼び出し元に戻す。行862では、MyServer型のメソッドfunc2が、long型の参照型とchar型のポインタ型の入力引数を取り、戻り値のない関数であることを宣言する。行864は、サーバと通信するための通信パケットbufを新たにメモリ割り当てする。行865で、bufに、long型の値を格納するメソッドpacklongを用いてkeyの値をbufに格納する。同様に行866で、bufに、文字列を格納するメソッドpackStringを用いて、valueが指す文字列をbufに格納する。行867で、callメソッドを用いて、サーバと通信を行う。行868は、サーバからの返答からlong型の出力引数を取り出し、keyに代入する。この際、Buffer型の提供するunpacklongを用いる。行869は不用になった通信パケットbufのメモリを開放する。行870で、制御を呼び出し元に戻す。

serverstub.c 900は、intf.h 800で宣言されたMyServer型の2つのメソッドをよびだすためのコードであるサーバループを格

納する。サーバループは、サーバが用いるもので、RPCの呼び出しを受ける側の

コードであり、無限ループの中でクライアントからのRPCを待ち、RPCの要求が到着したら、要求されたRPCの種類を見分け、適切な遠隔手続きの本体を呼び出す。サーバのプログラマは、`func1`および`func2`のソースコードを作成し、`serverstub.c 900`とあわせてコンパイルおよびリンクすることによって、サーバの実行可能プログラムを得ることができる。詳細は以下の通りである。

行901は、MyServer型の宣言を得るため、`intf.h 800`を読み込む。行902から行928がサーバループの定義である。行902で、サーバループ`loop`を引数なし、返回值なしの関数として宣言する。行904から927までが無限ループである。行905は通信パケット`buf`の宣言、行906はクライアントの通信情報（IPアドレス、ポート番号、通信コネクション、ユーザ情報等）を格納する型の宣言である。行907で、クライアントの`call`によって到着するRPC要求を待って停止する。あるクライアントのRPC要求が受け付けられると、該クライアントの情報が新たに割り当てられた`Client`型オブジェクトに、また新たにメモリ割り当てされた`Buffer`型オブジェクトに、`call`の第1引数および第2引数、すなわち呼び出すべき遠隔手続きの名前と、入力引数が格納される。これらのオブジェクトはそれぞれ、`client`および`buf`から指される。次に、行908で、もし起動すべき遠隔手続きの名前が`func1`であれば、処理に必要なローカル変数を定義し（行909）、`func1`の入力引数である`int`型データが`buf`に格納されているので、該データを取り出し（行910）、`func1`を呼び出す（行911）。`func1`の返回值は、再び`buf`に詰める（行912）。また、行913で、もし起動すべき遠隔手続きの名前が`func2`であれば、処理に必要なローカル変数を定義し（行914、行915）、`func2`の入力引数である`long`型データを取り出し（行916）、続いて`func2`の入力引数である`char`型へのポインタ型データを取り出す（行917）。これら2つの入力引数は、行918で`func2`を呼び出す際の引数に用いられる。行919では、出力引数である`key`の値を再び`buf`に詰める。また、行920で、起動すべき遠隔手続きの名前が`func1`でも`func2`でもなければ、エラーをクライアントに送信し（行921）、無限ループを繰り返す（行92

2)。最後に行924では、該クライアントに対し返事を送出し、続く行925および行926で不要になったデータを解放する。

以上が従来の遠隔手続き呼び出しのコンパイルを行った場合のIDLコンパイラ304の入力および出力の例である。

`intf'.idl` 1000は、RPCオブティマイザ101で変更されたIDLソースコードの例である。図1における新IDLソースコード106に対応する。行1001—1006がソースコード本体である。行1001は、RPCのインタフェース群をひとまとめにして、`MyServer`という名前をつけて宣言している。このIDLソースコードを用いて定義されるサーバオブジェクトは、`MyServer`という型を持つ。行1002および1003は、`intf.idl` 700と同じ`func1`および`func2`の宣言である。行1004および1005は、RPCオブティマイザ101が最適化のために追加した2つの遠隔手続きの宣言である。`func3`の引数は入力引数であり出力引数である`int`型変数`count`で、戻り値はない。`func4`の引数は入力引数である`int`型変数`i`で、戻り値はない。

`clientstub'.c` 1010は、RPCオブティマイザ101で変更されたクライアントのソースコードである。図1における新クライアントソースコード105、105'、…に対応する。記述言語はC++言語を用いている。行1011で、`intf'.idl` 1000から生成されるIDLヘッダファイルを読み込む。行1012が該クライアントの始まりである`main`関数の定義開始であり、行1019までが`main`関数の定義本体である。この定義では、説明を分かりやすくする目的で、いくつかのライブラリ関数を用いて例を示す。行1014は、該クライアントが、名前サービスまたは辞書から探し出すライブラリ関数を利用して、利用するサーバ探す処理である。”`MyServer`”という名前を指定して`lookupDirectory`ライブラリ関数を呼び出すことで、`MyServer`型のサーバを得ている。行1015はローカル変数`count`の定義および0への初期化である。行1016は、`MyServer`型のサーバ`server`の遠隔手続きである`func3`を呼び出す。ここでは、`client.c` 750で入力引数`i`の値を0から100まで変化させつつ、100回

のRPCを行っていた繰り返し処理を、1つのRPCにまとめあげて最適化している。行1017はcountの値を出力するライブラリprintf関数である。この行はclient.c 750と変化ない。行1018は、クライアントソースコード102、102'、…でfunc2およびfunc1の呼び出しを行っていた部分を、func4への1回のRPCに変更している。以上が最適化後の該クライアントの処理である。該クライアントでは、合計2回のRPCが実行される。

server+.c 1030は、RPCオブティマイザ101で生成されたサーバの追加ソースコードである。図1における追加サーバソースコード107に対応する。server+.c 1030は、サーバオブジェクトの既存のソースコードと、RPCオブティマイザ101が追加した遠隔手続きのインタフェースとの間のつなぎのソースコードを格納する。行1031は、intf'.idl 1000から生成されるIDLヘッダファイルを読み込む。行1032から1036はfunc3の定義、行1037から1041はfunc4の定義である。func1およびfunc2の定義は、従来通りサーバのプログラマを作成する。行1032は、関数func3がint型への参照型をとり、返り値なしであることを宣言する。行1034と1035は、ローカル変数iを用いてfunc1への呼び出しを100回繰り返し行い、返り値の合計をcountに保存している。ローカル変数iは、client.c 750中ではfor文以降使用されないのに、出力引数扱いにはならない。もしclient.c 750中でfor文以降にもiが使用されていれば、func3の変数iの最後の値は出力引数としてクライアントに返される。行1037はint型の入力引数を取り、返り値のない関数func4を宣言している。行1039でfunc2を呼び、行1040でfunc1を呼ぶ。func2の引数は、クライアントソースコード102、102'、…で使用されていた定数を埋め込んである。

以上が、RPCオブティマイザ101によってRPCを最適化した結果の例であるintf'.idl 1000、clientstub'.c 1010、server+.c 1030の例である。

内部処理フロー

次に、RPCオブティマイザ101の内部処理の詳細を述べる。RPCオブテ

イマイザ101は、図5を用いてすでに述べたように、レキシカルアナライザ503、パーザ504、および内部表現生成部505が、パース結果520、命令表521、基本ブロック表522、複合ブロック表523、および環境表524を作成し、また、IDLレキシカルアナライザ506、IDLパーザ507およびRPC表生成部508がRPC表527を作成する。中間コード変換部509は、作成されたこれらの表を用いてRPCの最適化を行い、その課程でサーバ側命令表525とサーバ側パース結果526が作成・変更されていく。そして最後に、ソースコード生成部510とIDLソースコード生成部511が、出力である新クライアントソースコード512、512'、…、追加サーバソースコード513、および新IDLソースコード514をパース結果520、命令表521、基本ブロック表522、複合ブロック表523、環境表524、RPC表527、サーバ側命令表525、およびサーバ側パース結果526から生成する。これらの内部処理のうち、RPCオプティマイザ101で特に重要となる中間コード変換部509の動作の詳細について以下に説明する。なお、以下では説明を簡単にするため、環境が入れ子にならない場合について説明するが、これは本発明の範囲を限定するものではない。

中間コード変換部509は、命令表521の基本命令列うち、RPCが多発する第1の基本命令列を検出するが、すでに述べたように本実施例ではこの検出に2つの方法を用いる。第1の方法はループによって繰り返し実行されるRPCを検出する方法、そして第2の方法は、基本ブロック内の一部または全部の基本命令列の中に登場するRPCの数が2以上である部分を検出する方法である。

図11を用いて第1の方法での処理の手順を説明する。

ステップ1101では、RPCを含む基本ブロック（以下Bと記す）を1つ選択する。これは、基本ブロック表522をから基本ブロックを1つ選択し、該基本ブロックの開始命令ID613から終端命令ID614に至る命令表521の命令表要素601のおのおのについて、以下の検査を行う処理である。該命令表要素の命令604が「手続き呼び出し」である基本命令を取り出し、ターゲット603、すなわち呼び出すべき手続き名を第1の手続き名とする。そして、RPC表527のRPC表要素651の各々について、該RPC表要素のRPC名652を調べ

、該RPC名が第1の手続き名と等しい場合、該基本ブロックはRPCを含むことが分かる。また、基本ブロック表522を順に選択していくには、各基本ブロックの次基本ブロック615と前基本ブロック616を用いて、木構造をたどる処理（トラバース）を行う。この手順で基本ブロックBを得る。

ステップ1102では、基本ブロックBを含む最内周ループを検知する。基本ブロック表522に格納された基本ブロック群は、次基本ブロック615で有向グラフを形成している。該有向グラフからループを検出する方法はすでに知られており、引用文献3（pp.445）などに示されているのでここでは特に説明しない。基本ブロックBを含む最内周ループは、検出されたループのうち、最も辺の数が少ないループを指す。

ステップ1103では、最内周ループの基本命令列を検査し、分離可能かどうかを判定する。判定がYESであれば（1104）、ステップ1106を行う。また判定がNOであれば（1105）、ステップ1125を行う。分離不可能なのは、ループ内にRPC以外の関数呼び出しを含む場合である。

ステップ1106では、最内周ループを手続きFに分離する。まず手続きFのために、新たな複合ブロック表要素を初期化する。複合ブロック表523に新たに第1の複合ブロック表要素を追加し、第1の複合ブロック表要素の複合ブロックID632に既存の複合ブロックと重複のない番号を割り当て、開始基本ブロックID633には前記最内周ループの先頭の基本ブロックのID、終端基本ブロックID634には該最内周ループの終端の基本ブロックのIDをそれぞれ格納する。また環境表524を新たに割り当て、第1の環境表とする。該第1の環境表の環境ID641には他の環境表と重複しない番号を割り当てる。親環境ID642と属性643には、該最内周ループに含まれるすべての基本ブロックの環境ID617の属性643をコピーする。第1の複合ブロック要素の環境ID635には、第1の環境表の環境ID641を格納する。第1の環境表の環境内変数表644には何も格納しない。

さらに、手続きFのために、新たなRPC表要素を割り当て、初期化する。RPC表527に新たなRPC表要素651である第1のRPC表要素を割り当てる

。第1のRPC表要素のPRC名652には、RPC表527の他のRPC表要素のRPC名と重複しない名前を生成し、割り当てる。IN引数表653とOUT引数表654には、現時点では何も格納しない。属性655には「自動生成」を格納する。

また、手続きFのために、新たに第2の基本ブロック表要素を割り当て、初期化する。この第2の基本ブロック要素は、前記最内周ループを手続きFとして取り出してサーバ側の手続きとした際に、手続きFを呼び出す命令列を保持するために用いる。第2の基本ブロック要素の基本ブロックID612には、他の基本ブロックと重複しない番号を割り当てる。開始命令ID613、終端命令ID614、次基本ブロック615、前基本ブロック616には、この時点ではなにも格納しない。また、環境表524を新たに割り当て、第2の環境表とする。該第2の環境表の環境ID641には他の環境表と重複しない番号を割り当てる。親環境ID642には「空」を格納する。属性643には「空」を格納する。第2の環境表の環境内変数表644には何も格納しない。第2の基本ブロック要素の環境ID635には、第2の環境表の環境ID641を格納する。さらに、第2の基本ブロック要素のDGEN変数表618、DKILL変数表619、DIN変数表620、DOUT変数表621、LIN変数表622、LOUT変数表623、LUSE変数表624、およびLDEF変数表625を以下のように初期化する。なお、ここで \cup を和集合、 \cap を集合の共通部分、 $-$ を集合の引き算とする。優先順位は設けず、左から右に順に計算する。操作対象はいずれも変数表であるので、例えば $A \cup B$ は、変数表Aと変数表Bに含まれるすべての変数表要素をあわせて新たな変数表を作り、その中から変数名662が重複する変数表要素を削除した変数表を得る操作である。また、 B_i のもつ8種の変数表を、 $B_i.DGEN$ 、 $B_i.DKILL$ 、 $B_i.DIN$ 、 $B_i.DOUT$ 、 $B_i.LIN$ 、 $B_i.LOUT$ 、 $B_i.LUSE$ 、 $B_i.LDEF$ と表わす。さらに前記最内周ループに含まれる基本ブロックを、 B_1 、 B_2 、 \dots 、 B_n とし、ループの先頭を B_1 とする。第2の基本ブロックのDGEN変数表618は、 $B_1.DGEN \cup B_2.DGEN \cup \dots \cup B_n.DGEN$ と設定する。また、 B_1 、 B_2 、 \dots 、 B_n を該最内周ループで実行しうる順に並べた列を S_1 、 S_2 、 \dots 、 S_m とする。たとえば S_i が B_a 、 B_b 、 \dots 、 B_z であるとし、 F_i を B_a 、

DKILLUBb.KILLU...UBz.DKILLとすると、第2の基本ブロックのDKILL変数表619を $F1 \cap F2 \cap \dots \cap Fm$ と設定する。そして、第2の基本ブロックのDIN変数表620は $B1.DIN$ に、また第2の基本ブロックのDOUT変数表621は $(DIN変数表620 - DKILL変数表619) \cup DOUT変数表621$ に設定する。また、LDEF変数表625は、上記 Si について、 Gi を $Ba.LDEF \cup Bb.LDEF \cup \dots \cup Bz.LDEF$ とすると、LDEF変数表625は $G1 \cap G1 \cap \dots \cap Gm$ と設定する。LUSE変数表624は、 $B1.LUSE \cup B2.LUSE \cup \dots \cup Bn.LUSE$ に設定する。また、前記最内周ループ内の基本ブロック群から制御が渡りうる該最内周ループ外の基本ブロック群 $C1, C2, \dots, Ck$ とすると、LOUT変数表623を $C1.LIN \cup C2.LIN \cup \dots \cup Ck.LIN$ とする。また、LIN変数表622は $B1.LIN$ とする。

ステップ1107では、既に検査した基本ブロックであるかを検査する。すでに検査した基本ブロックであるかどうかは、基本ブロック表610の環境ID617の属性643に、「検査済み」というタグが入っているかどうかで分かる。判定がYESであれば(1108)、ステップ1125を行う。また判定がNOであれば(1109)、ステップ1110を行う。

ステップ1110では、Fが危険な変数参照を行う可能性があるかを検査する。

判定がYESであれば(1111)、ステップ1113を行う。また判定がNOであれば(1112)、ステップ1117を行う。代表的な危険な変数参照・変更は、同じ変数に別の名前がつけられたエイリアスによる参照・変更である。例えばint*型(int型へのポインタ型)の変数pとqがあった場合、*p(ポインタpが指す先)と*qは、一般には異なるが、場合によっては同じint型のデータ領域である場合がある。このように同じデータを別の名前で参照・変更できる場合をエイリアスによる参照・変更と呼ぶ。

RPCのまとめあげを行う場合で、ローカル変数への代入命令も合わせてまとめあげる場合には、もともとクライアント側で行われていたローカル変数への代入がサーバ側で行われ、RPCの終了時に出力引数としてクライアント側に書き戻さ

れる。先ほどの`*p`への代入と`*q`への代入をサーバ側で実行しようとする、出力変数として2つの`int`型データ領域をサーバ側に用意し、これらをそれぞれ`p`および`q`で指した上でサーバ側のコードを実行し、しかる後に該2つの`int`型データ領域をクライアント側に送り返して、クライアント側の`*p`および`*q`に代入する、という手順をとる。

このため、`*p`と`*q`が互いにエイリアスである場合には、`*p`と`*q`への書き戻しの順番が問題となり、正しい計算が行えない場合がある。またサーバ側で、`*p`と`*q`への参照と代入がそれぞれ行われた場合には、`p`と`q`が同じデータを指しているか否かで計算結果が変化しうる。すなわち、このような危険な変数参照・変更が起きないようにするか、起きる危険が排除できない場合には該当部分のRCPのまとめあげを中止する必要がある。

危険な変数参照・変更の判定は、クライアントソースコードが利用しているプログラミング言語に依存するが、一般には変数の型を用いて以下の手順で行う。手続きFの外から入ってくる変数の集合(`Fin`とする)は前記第2の基本ブロックのDIN変数表620で、また手続きFの中で参照・変更される変数の集合(`Refer`とする)は第2の基本ブロックのLUSE変数表624に保存されている。さらに手続きFで変更される変数の集合(`Fassign`とする)は第2の基本ブロックのLUSE変数表624に保存されている。そこで、`Fassign`の各変数について、型を調べ、その型が他の変数のエイリアスになりうるかを検査する。ある型の変数がべつの変数のエイリアスになりうるのは間接参照・変更が可能な型に限られる。例えばC++言語の場合、ポインタ型の参照・変更、参照型、配列要素の3種類が該当する。また、Java言語の場合、同じクラスのオブジェクト型、または継承可能な親子関係を持つ2つのクラスのオブジェクト型が該当する。Java言語の場合には、型が厳密であるため、型の同一性および継承関係を信頼してよい。一方C++のように型をプログラマが強制変更できるプログラミング言語では、型の同一性および継承関係は、一般には役に立たない。この場合、あらゆるポインタ型、参照型、配列参照が、他の変数のエイリアスになりうる。検査の手順は、`Fassign`∪`Refer`に含まれる各変数について、上記の条件に鑑みてお互いにエイリアスになりうるかを決定し、エイリアスになりうる変数群を計

算する。エイリアスになりうる変数群が空でなければ、危険ありと判定する。

ステップ1113では、手続きFの危険な変数参照・変更が回避可能かを判定する。判定がYESであれば(1115)、ステップ1116を行う。また判定がNOであれば(1114)、ステップ1125を行う。危険な変数参照・変更が回避可能かどうかの判定は、エイリアスになりうる変数群の数が一定以下か否かである。例えばC++言語のポインタ型変数vの場合、vが前記F a s s i g nに含まれておらず、またエイリアスによる変更の可能性もなければ、vは手続きFの間1つのデータ領域を指していることが保証できる。また、J a v a言語の場合、オブジェクト型変数oが前記F a s s i g nに含まれていなければ、oは手続きFの間1つのデータ領域を指していることが保証できる。このように、エイリアスになりうる変数群がすべて、手続きFの間1つのデータ領域を指していることが保証でき、かつエイリアスになりうる変数群の数が一定以下ならば、回避可能と判定する。

ステップ1116では、手続きFの危険な変数参照・変更を回避するコードを追加する。エイリアスになりうる変数群のうち、互いにエイリアスになりうる2つの変数v1とv2について、「もしv1とv2が異なれば手続きFを(サーバ側で)実行する。そうでなければ、従来の基本命令列を(クライアント側で)実行する。」という基本命令列を、命令表521中、前記第1の複合ブロック表要素の開始基本ブロックID633で指される基本ブロックの、開始命令ID613の前に挿入する。

ステップ1117では、手続きFの環境を設定する。具体的には、前記第2の環境表の環境内変数表644に、前記第2の基本ブロック要素のL U S E変数表624に格納されている変数表を代入する。

ステップ1118では、手続きFの入力引数を設定する。具体的には、前記第2の基本ブロック要素のL U S E変数表624を、第1のR P C表要素のI N引数表653に代入する。

ステップ1119では、手続きFの出力引数を設定する。具体的には、前記第2の基本ブロック要素のD G E N変数表618∩L U S E変数表624を計算し、第1のR P C表要素のO U T引数表654に代入する。

ステップ1120では、手続きFの通信コストを計算する。前記ステップ11

18 およびステップ1119で計算したIN引数表653とOUT引数表654に含まれる変数群について、型663からサイズを得る。なお、このサイズは、型によっては実行時まで不明な場合があることに注意が必要である。このような場合の型のサイズは考える最小サイズとする。これらサイズを合計し、手続きFの起動時および終了時に必要な総転送データ量を得る。

ステップ1121では、上記総転送データ量が定数以上であるかを判定する。判定がYESであれば(1122)、ステップ1125を行う。また判定がNOであれば(1123)、ステップ1124を行う。

ステップ1124では、手続きFを採用し、データ構造に登録する。具体的には、前記最内周ループを構成する基本ブロック群、すなわち第1の複合ブロック表要素の開始基本ブロックID633から終端基本ブロックID634に至る基本ブロック群の各々について、開始命令ID613と終端命令ID614の間の基本命令列を、サーバ側命令表525に移動する。また、該基本命令列に対応するパーズ結果520の一部を、サーバ側パーズ結果526に移動する。また、「手続きFの入力引数をスタックに積み、手続きFを呼び出し、手続きFの出力引数をスタックから下ろす」という基本命令列を、命令表521中、前記第1の複合ブロック表要素の開始基本ブロックID633で指される基本ブロックの、開始命令ID613の前に挿入し、この基本命令列の先頭と終端を前記第2の基本ブロック要素の開始命令ID613と終端命令ID614で指す。また、第2の基本ブロック要素の次基本ブロック615に、前記第1の複合ブロック表要素の開始命令ID613で指される基本ブロックの前基本ブロック616を設定する。

さらに第2の基本ブロック要素の次基本ブロック615に、前記第1の複合ブロック表要素の終端基本ブロックID634で指される基本ブロックの次基本ブロック615を設定する。そして、第1の基本ブロックの環境ID617の属性643に、「検査済み」というタグを追加する。

ステップ1125では、手続きFを破棄し、一時的に作ったデータ構造群を解放する。具体的には、前記第1の複合表要素、前記第1の環境表、前記第2の基本ブロック表要素、および前記第2の環境表を解放し、第1の基本ブロックの環境ID617の属性643に、「検査済み」というタグを追加する。

ステップ1126では、すべての可能性を調べたかを検査する。すなわち、基本ブロック表522のすべての基本ブロック表要素について、環境ID617の属性643に、「検査済み」というタグがあるかどうかを検査する。判定がYESであれば(1128)、RPCを含むループを最適化する処理を終了する。また判定がNOであれば(1127)、ステップ1101にもどる。

以上がRPCを含むループを最適化する処理の手順である。

次に、図12を用いて第2の方法の処理の手順を説明する。

ステップ1201では、RPCを1つ選択する。これは、基本ブロック表522を基本ブロックを1つ選択し、該基本ブロックの開始命令ID613から終端命令ID614に至る命令表521の命令表要素601のおののについて以下の検査を行う。該命令表要素の命令604が「手続き呼び出し」である基本命令を取り出し、ターゲット603、すなわち呼び出すべき手続き名を第1の手続き名とする。そして、RPC表527のPRC表要素651の各々について、該RPC表要素のPRC名652を調べ、該RPC名が第1の手続き名と等しい場合、該命令表要素はRPCである。該命令表要素をSと呼ぶ。また、Sを含む基本ブロックを、第1の基本ブロックと呼ぶ。さらに該基本ブロックの命令表要素を検査していき、次のRPCが命令表要素に現れた場合に、Sを採用し、ステップ1202以下の処理を行う。もしそのような次のRPCが第1の基本ブロック内にない場合、次の基本ブロックを検査する。

ステップ1202では、Sに続くRPCであるTを得る。これは、Sから、前記第1の基本ブロックの終端命令ID614に至る命令表521の命令表要素601を順に検査し、RPCを検出する。ステップ1201の処理から、このようなRPCが検出できることは保証される。検出されたSに続くRPCをTと呼ぶ。

ステップ1203では、SとTの間の基本命令列が分離可能かを判定する。判定の結果がYESであれば(1205)、ステップ1206を行う。一方判定の結果がNOであれば(1204)、ステップ1224を行う。分離不可能なのは、SからTに至る基本命令列中にRPC以外の関数呼び出しを含む場合である。

ステップ1206では、SとTの間の基本命令列を手続きF'として分離する。

。

このステップの処理は、既に説明したステップ1106の処理と同等であるため、繰り返して説明しない。

ステップ1207では、既に検査した基本ブロックであるかを検査する。すでに検査した基本ブロックであるかどうかは、基本ブロック表610の環境ID617の属性643に、「Sは検査済み」というタグが入っているかどうかで分かる。判定がYESであれば(1208)、ステップ1224を行う。また判定がNOであれば(1209)、ステップ1210を行う。

ステップ1210では、手続きF'が危険な変数参照・変更を行う可能性があるかを検査する。判定の結果がYESであれば(1211)、ステップ1213を行う。一方判定の結果がNOであれば(1212)、既に説明したステップ1110の処理と同等のステップ1217を行う。

ステップ1213では、手続きF'の危険な変数参照・変更が回避可能かを判定する。判定の結果がYESであれば(1215)、ステップ1216を行う。一方判定の結果がNOであれば(1214)、既に説明したステップ1113の処理と同等のステップ1224を行う。

ステップ1216では、手続きF'の危険な変数参照・変更を回避するコードを追加する。このステップの処理は、既に説明したステップ1116の処理と同等であるため、繰り返して説明しない。

ステップ1217では、手続きF'の環境を設定する。このステップの処理は、既に説明したステップ1117の処理と同等であるため、繰り返して説明しない。

ステップ1218では、手続きF'の入力引数を設定する。このステップの処理は、既に説明したステップ1118の処理と同等であるため、繰り返して説明しない。

ステップ1219では、手続きF'の出力引数を設定する。このステップの処理は、既に説明したステップ1119の処理と同等であるため、繰り返して説明しない。

ステップ1220では、手続きF'の通信コストを計算する。このステップの処理は、既に説明したステップ1120の処理と同等であるため、繰り返して説明

しない。

ステップ1221では、ステップ1220で計算した総転送データ量が定数以上であるかを判定する。判定の結果がYESであれば(1222)、ステップ1224を行う。一方、判定の結果がNOであれば(1223)、ステップ1225を行う。

ステップ1224では、手続きF'を破棄し、一時的に作ったデータ構造群を解放する。このステップの処理は、既に説明したステップ1125の処理と同等であるため、繰り返して説明しない。ただし、現在処理中のSに関して、すでにステップ1225を通過している場合、すなわちさらなるまとめあげの可能性を探っている場合には、手続きF'を破棄せずに、後述するステップ1225と同じ処理を行って手続きF'を採用する。また、前記第1の基本ブロックに追加するタグは、「Sを検査済み」とする。

ステップ1225では、手続きF'の次のRPC(Uとする)を得て、手続きF'とUがまとめあげ可能かどうかを判定する。判定の結果がYESであれば(1226)、SをTに置き換えて、ステップ1202を行う。この場合には、続くステップ1206で新たな手続きF'を、S、T、Uを含む命令列から作成する。一方判定の結果がNOであれば(1227)、ステップ1228を行う。

ステップ1228では、手続きF'を採用し、データ構造に登録する。このステップの処理は、既に説明したステップ1124の処理と同等であるため、繰り返して説明しない。ただし、前記第1の基本ブロックに追加するタグは、「Sを検査済み」とする。

ステップ1229では、すべての可能性を調べたかを検査する。判定の結果がYESであれば(1231)、RPCが多発する命令列を最適化する処理を終了する。一方判定の結果がNOであれば(1230)、ステップ1201を行う。すなわち、命令表521のすべてのRPC Vについて、「Vは検査済み」のタグがいずれかの基本ブロックについていれば、判定はYESである。判定がYESであれば(1128)、RPCを含むループを最適化する処理を終了する。また判定がNOであれば(1127)、ステップ1101にもどる。

以上がRPCが多発する命令列を最適化する処理の手順である。

サーバ側処理

動的リンカ 1 1 6 が追加サーバオブジェクト 1 1 5 とサーバオブジェクト 1 0 4 を結合し、最適化された R P C の効果を得るためには、クライアントから送られる要求を追加サーバオブジェクト 1 1 5 とサーバオブジェクト 1 0 4 の間で適切に振り分ける方法、または追加サーバオブジェクト 1 1 5 に送られてきた要求の一部をサーバオブジェクト 1 0 4 に送る方法が必要となる。以下に、この実現方法について 3 つ説明する。特に、サーバオブジェクト 1 0 4 はパッケージとして売られており、すでにコンパイル済みで、ソースコードの入手が困難である場合があることに注意が必要である。

第 1 の方法として、拡張可能なディスパッチャを用いる方法を、図 1 3 および図 1 4 を用いて説明する。ディスパッチャとは、サーバの一部として組み込まれるモジュールで、クライアントからの要求を解析し、適切な遠隔手続きを呼び出す一連のコードである。ディスパッチャの例としては、図 9 で示したサーバスタブがある。図 9 のサーバスタブは、呼び出すことができる遠隔手続きの種類が `f u n c 1` と `f u n c 2` の 2 種類にかぎられていた。しかし、9 0 8 から 9 2 0 の間の `i f` 文による分岐を、図 9 のようなハードコードされた分岐ではなく、テーブル駆動の分岐列として実現することもできる。ここでテーブルとは、「クライアントが指定する遠隔手続きのキー（図 9 の例では” `f u n c 1` ” や” `f u n c 2` ” の文字列）、遠隔手続きの引数を操作する手続き、遠隔手続き本体」という 3 つ組を要素とする。クライアントの要求に対して、このテーブルの要素を一つ一つチェックし、キーが合う要素があれば、引数操作手続きと遠隔手続き本体を起動する、という処理を行う。

ディスパッチャを拡張可能にすることは、上記テーブルに要素を追加可能にすることで実現できる。図 1 3 と図 1 4 に、ディスパッチャを拡張する前と後の動作の違いを模式的に図で表す。

サーバオブジェクト 1 3 0 1 は、最適化前のサーバオブジェクトである。該サーバオブジェクトには、あらかじめ拡張可能ディスパッチャ 1 3 0 3 が組み込まれている。クライアントからの遠隔手続き呼び出しは通信ポート 1 3 0 2 に格納され

、拡張可能ディスパッチャ1303が順次、遠隔手続き呼び出しを読み出して解析する(1350)。ディスパッチャに登録されている遠隔手続きはfunc1 1304とfunc2 1305であり、クライアントからの要求がこれらの遠隔手続きへの呼び出し要求であれば、拡張可能ディスパッチャ1303は要求を処理する(1351、1352)。これらの遠隔手続き以外への呼び出しはエラーとなる。

一方、RPCオブティマイザ101による最適化によって、サーバオブジェクト(server.exe)に追加サーバオブジェクト(server+.exe)が組み込まれると、図14のようになる。サーバオブジェクト1401の拡張可能ディスパッチャ1403は、通信ポート1402からクライアントの遠隔手続き呼び出し要求を取り出して解析する(1450)。拡張後の拡張可能ディスパッチャ1403には、4つの遠隔手続き(func1 1404、func2 1405、func2 1406、func2 1407)が登録される。この登録は、前記テーブルへの要素の登録によって実現され、func2 1406、func2 1407のサーバオブジェクト1401への動的リンクと同時に動的リンカ116によって行うことができる。この結果、クライアントからの4種類の遠隔手続き呼び出しに応え、func1 1404、func2 1405、func2 1406、func2 1407を呼び出す(1451、1452、1453、および1454)ことが可能となる。

サーバオブジェクト104と追加サーバオブジェクト115を結合させる第2の方法は、図15で示すように、ディスパッチャにインタプリタ型言語を内蔵する方法である。動的リンカ116は、追加サーバオブジェクト115をサーバオブジェクト1501と同じアドレス空間中に送り込むことはできる。このため、いかにディスパッチャに入ってくる要求を追加サーバオブジェクト115に渡すかが問題になるが、図15ではディスパッチャにインタプリタを組み込んでおき、追加サーバオブジェクト115をサーバオブジェクト104に結合する際に、該インタプリタの実行するプログラムに「func3およびfunc4の要求は、追加サーバオブジェクト115に渡せ」という制御を付け加える。付け加える作業は該インタプリタが解釈実行しているプログラムを変更することにあたるため

、問題なく可能である。また、必要ならば変更すべきプログラムをネットワーク経由でサーバオブジェクト1501に送ることも可能である。この結果、インタプリタ付きディスパッチャ1503は通信ポート1502から受けとった要求が

(1550)、func3およびfunc4への要求であった場合、インタプリタで前記「func3およびfunc4の要求は、追加サーバオブジェクト115に渡せ」という処理を行う。一方”func1”または”func2”への要求は、従来どおりfunc1 1504またはfunc2 1505に渡され、処理される(1551、1552)。

サーバオブジェクト104と追加サーバオブジェクト115を結合させる第3の方法は、図16に示すように、一旦追加サーバオブジェクト115にすべての要求を与え、必要に応じてサーバオブジェクト104に要求を転送する方法である。この場合には、サーバオブジェクト104に対する動的リンクは必要ない。

クライアントからの要求は、まず追加サーバオブジェクト1601の通信ポート1605に格納され、ディスパッチャ1602によって解析される(1650)。この際、要求されたRPCが追加サーバオブジェクト1601の提供するfunc3 1603またはfunc4 1604であれば、これらのうち適切な遠隔手続きを起動する(1651、1652)。しかし、func3 1603およびfunc4 1604は、クライアントからの一連のRPCをまとめあげた遠隔手続きであるので、func1 1613やfunc2 1614への呼び出しも含む。このような呼び出しは、サーバオブジェクト1611の通信ポート1615に送られる(1653および1654)。通信ポート1615に格納された要求は、従来どおりディスパッチャ1612が解析し(1655)、func1 1613またはfunc2 1614への呼び出しが起動される(1656、1657)。また、通信ポート1605にfunc1 1613やfunc2 1614への要求が到着した場合には、ディスパッチャ1602は該要求を通信ポート1615に転送する(1658)。この方法では、サーバオブジェクト1611には一切の変更が不要である。なおかつ、追加サーバオブジェクト1601からサーバオブジェクト1611へのRPC(1653、1654、1658)は同一の計算機で行われるため、計算機間のRPCに比べてコストが非常に小さい。このため、RPC最適

化の効果は依然として高い。

静的・動的最適化

次に、図17を用いて、RPCオブティマイザ101の追加機能を説明する。計算機203'に通信計測部1701、または計算機203に通信計測部1702を備える。これら通信計測部は、実行中のクライアントオブジェクト114から新サーバオブジェクト117への各種RPCの頻度、および入力引数と出力引数のサイズを計測して記録する。記録された情報である実行時通信情報1703は、次のコンパイル時にRPCオブティマイザ101（図1）の変形であるRPCオブティマイザ1700に送られる。なお次のコンパイルは、クライアントオブジェクト114の開発者が手動で行ってもよいし、システム管理者が定期的に行ってもよいし、また、動的コンパイルによってクライアントオブジェクト114の動作中に必要に応じて行ってもよい。実行時通信情報1703は、RPCオブティマイザ1700が通信コストを評価する処理、すなわち前述のステップ1120やステップ1220で用いられる。実行時通信情報1703によって、引数のサイズが実行時まで不明な場合、たとえば引数の一部が可変長配列、可変長文字列、共用体等である場合に、ステップ1120やステップ1220の転送データ量の計算をより正確に行うことができる。

拡張IDL

RPCの最適化がよりよく行えるように、遠隔手続きの性質を詳しく記述する拡張IDLの説明を図18を用いて行う。遠隔手続きの性質とは、手続きによる副作用の有無、手続きと手続きの並列実行可能性、手続きと手続きの実行順序交換可能性等である。これらが分かると、まとめあげ方の選択肢が広がり、結果としてまとめあげの効率が上がる。このため、RPCオブティマイザは通常のIDLの他、拡張IDLも入力としてとる。

`extended intf.idl 1800`は拡張IDLを用いたRPCインタフェースの記述例である。行1801—1807でサーバオブジェクトの型を宣言し、行1802、1803、1804でそれぞれ`func1`、`func2`、`func3`のインタフェースを宣言している。行1802に、第1の拡張として`func1`が副作用を持たない遠隔手続きであることを示す宣言である`con`

st 予約語が付加される。また第2の拡張として、行1805で、func2とfunc3の実行順序が交換可能であることを宣言している。なお、副作用がないことは、同じサーバオブジェクトが提供する他のすべての遠隔手続きと、実行順序が交換可能であることを意味する。さらに第3の拡張として、行1806で、func1、func2とfunc3が並列実行可能であるということを宣言している。

これらの宣言は、P R C表650の属性655に格納される。これらの宣言の利用方法は種々考える。第1の宣言（副作用なし）および第2の宣言（実行順序が交換可能）を利用する1つの方法は、中間コード変換部509のステップ1202において、次にSに続くR P CであるTを得る部分で利用する方法である。前述したTの検出方法だけではなく、TのあとにTと実行順序が交換可能なR P C（Uとよぶ）がある場合には、Tの代わりにUを残りのステップに与える。これによって、SとTをまとめあげるだけでなく、SとUをまとめあげることも試みる。

また、第3の宣言（並列実行可能）の利用方法も種々考えるが、利用方法の1つは、ソースコード生成部510が追加サーバソースコード107を生成する際に、複数の遠隔手続きをシーケンシャルに呼び出すコードを生成するかわりに、複数の遠隔手続きを並列して呼び出すコードを生成することである。例えば、`func1`、`func2`が並列実行可能であることを利用すれば、追加サーバソースコード`server+.c 1030`を1820に示す別の追加サーバソースコードを出力する。

`server+.c` 1820は並列実行可能性を加味した追加サーバソースコードの例である。行1821は、`int f'.idl 1000`から生成されるIDLヘッダファイルを読み込む。行1822は、遠隔手続きの並列実行に必要なスレッドライブラリのヘッダファイルを読み込む。ここで、スレッドとは複数並列に動作可能な実行単位の一つ一つを指す。行1823から1836は`func 3`の定義、行1837から1847は`func 4`の定義である。`func 1`および`func 2`の定義は、従来通りサーバのプログラマを作成する。行1823は、関数`func 3`が`int`型への参照型をとり、返り値なしであることを宣言する。行1825、1826、1827は、スレッドの識別子を格納するためのリスト構造`a1`

`lThreads`、スレッドのために必要なローカル変数配列`lt`、およびスレッドの
 戻り値へのローカル変数`rval`の宣言である。行1828から1831は、`func1`への呼び出しを、
 各々新たなスレッドで行う。1832から1835は、各スレッドから`func1`の戻り値を得て、`count`に合計する。
`rval`は`func1`の戻り値へのポインタである。スレッドは複数並列に実行されるので、
 1820における`func1`に対する100回の呼び出しは、並列実行される。特に並列機上では並列に実行される可能性が高い。

同様に、行1837は、関数func4の型宣言である。行1839と1840はスレッドの識別子を格納するためのリスト構造allThreadsとスレッドのために必要なローカル変数配列tの宣言である。行1841から行1844でfunc2およびfunc1を2つのスレッドで実行する。行1845と行1846で、上記2つのスレッドが終了したのを確認して、func4を終了する。このコードによって、func2とfunc1が並列実行される。

上記の例で分かる通りスレッド実行のためのコード生成は、比較的機械的なテンプレートにより実現することが可能である。また、一旦どの処理部分を並列に実行するかがわかれば、このようなスレッド実行のためのコード生成を行うコンパイラは、すでに多く知られているため、このようなコード生成の詳細ここでは説明しない。

第1の変形例

さて、図1ではR P Cオブティマイザ101はクライアントソースコード102、102'、…とI D Lソースコード103を入力とし、新クライアントソースコード105、105'、…、新I D Lソースコード106、追加サーバソースコード107を出力するが、R P Cオブティマイザ101の機能に本質的な変更を加えずに、他の状況に対応する本発明の変形例を作ることとも可能である。

図19を用いて、図3で示した従来の遠隔手続き呼び出しの環境に対して好適な本発明の実施形態の例（第1の変形例）を説明する。

図19は、クライアントのコンパイル時から、クライアントとサーバの実行時
に至る、第1の変形例の利用の流れを示す構成図である。

RPCオブティマイザ1900は、クライアントを構成するために用いられる

クライアントソースコード301、301'、…と、サーバの提供する遠隔手続き群のインタフェースをIDLで記述したIDLソースコード302と、サーバを構成するために用いられるサーバソースコード303、303'、…とを入力としてとり(1940—1944)、RPCを最適化し、その結果、新クライアントソースコード1901、1901'、…(1945、1946)、新IDLソースコード1902(1947)、および新サーバソースコード1903、1903'、…(1948、1949)を出力として出す。

IDLコンパイラ1904は、新IDLソースコード1902を入力として動作し(1950)、クライアントスタブ1905(1951)、RPCヘッダファイル1906(1952)、およびサーバスタブ1907(1953)を出力する。

新クライアントソースコード1901、1901'、…とクライアントスタブ1905とRPCヘッダファイル1906は、コンパイラ・リンカ1908によってコンパイル及びリンクされ(1954、1955、1956、1957)、クライアントの実行可能プログラムであるクライアントオブジェクト1910となり出力される(1962)。また、新サーバソースコード1903、1903'、…とサーバスタブ1907とRPCヘッダファイル1906は、コンパイラ・リンカ1909によってコンパイル及びリンクされ(1958、1959、1960、1961)、サーバオブジェクト1911となり出力される(1963)。以上がコンパイル時の動作である。

図19に示す実行時の一例では、クライアントオブジェクト1910が計算機203'上で実行される(1964)。一方サーバオブジェクト1911は計算機203上で実行される(1965)。実行時に起こるRPC(1966)は、IDLソースコード302にもともと宣言されていた遠隔手続きと、新IDLソースコード1902で宣言された遠隔手続きのどちらか、または両方が起こりうる。

上記第1の変形例では、追加サーバソースコード107を新サーバソースコード1903、1903'、…の一部として出力する点が上記実施例と異なるが、図5を用いて説明したRPCオブティマイザ101の内部構造と同様の内部構造でRPCオブティマイザ1900が実現できる。このため、RPCオブティマイザ1

900の内部構造を改めて説明することはしない。

第2の変形例

図20を用いて、Java言語の遠隔手続き呼び出しであるRemote Method Invocation(RMI)に対して好適な本発明の実施形態の例(第2の変形例)を説明する。RMIでの、IDLコンパイラであるJava言語の中間コード形式であるclassファイルを入力としてとり、出力もclassファイルである。

図20は、クライアントのコンパイル時から、クライアントとサーバの実行時に至る、第2の変形例の利用の流れを示す構成図である。

RPCオブティマイザ2003は、クライアントを構成するために用いられるクライアントソースコード2001と、サーバの実行可能プログラムであるサーバクラスコード2002とを入力としてとり(2050、2051)、RPCを最適化し、その結果、新クライアントソースコード2004(2052)と追加サーバソースコード2005(2053)を出力として出す。なお図20ではソースコードとしてクライアント側、サーバ側各々1つのみ用いているが、これらが複数であっても差し支えない。サーバクラスコード2002は遠隔手続きのインタフェース定義を含んでいるので、RPCオブティマイザ2003は、図1ではIDLに宣言されているRPCインタフェースに関する情報を、RMIコンパイラと同様の手法でサーバクラスコード2002から抽出することができる。

新クライアントソースコード2004は、コンパイラ2006によって中間コード形式にコンパイルされ(2054)、Java仮想マシン(JVM)で実行可能なクライアントクラスコード2008となり出力される(2055)。また、追加サーバソースコード2005は、コンパイラ2007によって中間コード形式にコンパイルされ(2056)、Java仮想マシン(JVM)で実行可能な追加サーバクラスコード2009となり出力される(2057)。さらに追加サーバクラスコード2009は、RMIコンパイラ2010によってコンパイルされ(2058)、クライアントスタブ2011とクライアントスタブ2011が出力される(2059、2060)。以上がコンパイル時の動作である。

図20に示す実行時の一例では、クライアントクラスコード2008とクライ

アントスタブ2011が計算機203'上のJVM 2013で実行される(2061、2062)。一方サーバクラスコード2002と追加サーバクラスコード2009とサーバスタブ2012は、計算機203上のJVM 2014で実行される(2063、2064、2065)。実行時に起こるRPC(2066)は、サーバクラスコード2002にもともと宣言されていた遠隔手続きと、追加サーバソースコード2005で宣言された遠隔手続きのどちらか、または両方が起こりうる。この際の遠隔手続き呼び出しは、2067、2066、2068、2069の順に転送される。

第2の変形例では、図1の実施例ではIDLソースコード103によって得ていたRPCインタフェースに関する情報を、サーバクラスコード2002から抽出する点異なる。しかしこの抽出処理は、すでにRMIコンパイラが実施している実施している公知の方法と同じである。

また、第2の変形例ではJava仮想マシンがサーバオブジェクト(サーバクラスコード2002)が動作する計算機上に配置されている。この点と、Javaのクラスコードがネットワーク経由で送受信可能である点とを利用することによって、第2の変形例のRPCオプティマイザが生成する追加サーバクラスコード2009およびサーバスタブ2012を、サーバクラスコード2002が動作中・動作前にかかわらず、計算機203'から計算機203に送付することができる。なおこの送付の動作に関しては、Javaの基本機能の1つであるので、ここで繰り返して説明することはしない。前記実施例でも、サーバ側にインタプリタまたは言語の実行環境を装備することで、同様の効果を得ることができる。

第3の変形例

図21を用いて、Java言語の遠隔手続き呼び出しであるRemote Method Invocation(RMI)に対して好適な本発明の実施形態の別の例(第3の変形例)を説明する。

図21は、クライアントのコンパイル時から、クライアントとサーバの実行時に至る、第3の変形例の利用の流れを示す構成図である。

RPCオプティマイザ2103は、クライアントを構成するために用いられるクライアントソースコード2101と、サーバを構成するために用いられるサーバソ

スコード2102とを入力としてとり(2150、2151)、RPCを最適化し、その結果、新クライアントソースコード2104(2152)と新サーバソースコード2105(2153)を出力として出す。なおこの図ではソースコードをクライアント側、サーバ側各々1つのみ用いているが、これらが複数であっても差し支えない。サーバソースコード2102は遠隔手続きのインタフェース定義を含んでいるので、RPCオブティマイザ2103は、図1ではIDLに宣言されているRPCインタフェースに関する情報を、サーバソースコード2102から抽出することができる。

新クライアントソースコード2104は、コンパイラ2106によって中間コード形式にコンパイルされ(2154)、Java仮想マシン(JVM)で実行可能なクライアントクラスコード2108となり出力される(2155)。また、新サーバソースコード2105は、コンパイラ2107によって中間コード形式にコンパイルされ(2156)、Java仮想マシン(JVM)で実行可能なサーバクラスコード2109となり出力される(2157)。さらにサーバクラスコード2109は、RMIコンパイラ2110によってコンパイルされ(2158)、クライアントスタブ2111とクライアントスタブ2111が出力される(2159、2160)。以上がコンパイル時の動作である。

図21に示す実行時の一例では、クライアントクラスコード2108とクライアントスタブ2111が計算機203'上のJVM 2113で実行される(2161、2162)。一方サーバクラスコード2109とサーバスタブ2112は、計算機203上のJVM 2114で実行される(2163、2164)。実行時に起こるRPC(2166)は、サーバソースコード2102にもともと宣言されていた遠隔手続きと、新サーバソースコード2105で宣言された遠隔手続きのどちらか、または両方が起こりうる。この際の遠隔手続き呼び出しは、2165、2166、2167の順に転送される。

上記第3の変形例では、上記図1の実施例でIDLソースコード103によって得ていたRPCインタフェースに関する情報を、サーバソースコード2102から抽出する点が異なる。しかしこの抽出処理は、すでにJavaコンパイラおよびRMIコンパイラが実施している方法と同じであり、公知であるので、ここでは改

めて説明することはしない。

本発明によれば、以下の（a）から（g）の効果がある。

（a）サーバのみ、またはクライアントのみを改変することによりR P Cの最適化を行うのではなく、サーバとクライアントの両方を改変することにより、複数のR P Cを1回のR P Cにまとめることにより、サーバ・クライアントのR P Cを高速に実現する。

（b）クライアントのソースコードを解析してクライアントが複数の遠隔手続きを連続的に発行する実行列を検出し、該実行列を実行する1つの新たな遠隔手続きをサーバに追加することにより、クライアント側で頻発するR P Cを、1発のR P Cで処理するサーバに拡張可能となる。

（c）クライアントのソースコードを解析してクライアントが複数の遠隔手続きを連続的に発行する実行列を検出し、該実行列を1回のR P Cで実行する新たな遠隔手続きの引数および戻り値を決定し、該新たな遠隔手続きの引数および戻り値をI D L記述に追加することにより、クライアント側で頻発するR P Cをまとめて処理する新たな遠隔手続きをクライアントから呼び出し可能となる。

（d）主にI D L記述がサーバのソースコードと別に提供される場合のため、クライアントのソースコードとI D L記述を入力として、改変したクライアントのソースコードと、改変したI D L記述と、追加すべきサーバのソースコードを出力することにより、プログラマが書いた複数のR P Cを実行時には1回で処理するオブジェクトを作成可能となる。

（e）主にI D L記述がサーバのオブジェクトに（概念的に）埋め込まれている場合のため、クライアントのソースコードとサーバのオブジェクトを入力として、改変したクライアントのソースコードと、サーバに追加すべきソースコードを出力する。これにより、J a v a言語等で遠隔手続きの名前と型がサーバのソースコード中に記述されている場合に遠隔手続き呼び出しの最適化を行うことが可能となる。

（f）主にI D L記述がサーバのソースコードに（概念的に）埋め込まれている場合のため、上記最適化をプログラマのあらたな手間を生じさせずに行うため、クライアントのソースコードとサーバのオブジェクトを入力として、改変したクライ

アントのソースコードと、サーバに追加すべきソースコードを出力する。これにより、Java言語（JavaはSun Microsystems Inc.の登録商標である。）で遠隔手続きの名前と型がサーバのソースコード中に記述されている場合等に遠隔手続き呼び出しの最適化を行うことが可能となる。

（g）遠隔手続きの性質をIDL記述中に表現する方法を備えることにより、クライアントが複数の遠隔手続きを連続的に発行する実行列を検出した際に、該実行列を実行する1つの新たな遠隔手続きとすることを助けることが可能となる。

以上の効果により、一連のRPCをまとめあげ、これによりRPCにおいて遠隔手続きが多数呼び出される場合の性能を向上する。この結果、遠隔手続きを用いたプログラムの作成が容易になる。

特許請求の範囲

(THE INVENTION CLAIMED IS)

【請求項 1】

少なくとも 1 個のプログラムまたはプログラム部品を実行させる計算機上の、少なくとも 1 個の遠隔手続きを提供するサーバオブジェクト（以下、サーバ）と該遠隔手続きを呼び出す遠隔手続き呼び出し（以下、R P C）によって処理を進めるクライアントオブジェクト（以下、クライアント）との間で R P C を最適化する方法であって、

クライアントのうち R P C が複数回発生する処理部分を、該発生回数より少ない回数の R P C で実行するための新たな遠隔手続き及び該遠隔手続きのインターフェースをサーバに追加する処理と、
クライアントが該新たに追加した遠隔手続きを呼び出して該処理部分を該少ない回数の R P C で実行する処理と、
を有することを特徴とする遠隔手続き呼び出し最適化方法。

【請求項 2】

少なくとも 1 個のプログラムまたはプログラム部品を実行させる計算機上の、少なくとも 1 個の遠隔手続きを提供するサーバと該遠隔手続きを呼び出す R P C によって処理を進めるクライアントオブジェクトとの間で R P C を最適化する方法であって、

該クライアントのソースコードを解析することにより、連続して実行される可能性が高く R P C の並びである遠隔手続き実行列を検出する処理と、

該遠隔手続き実行列を実行する新たな遠隔手続きをサーバに追加する処理と、クライアントが該新たに追加した遠隔手続きを呼び出して該処理部分を該少ない回数の R P C で実行する処理と、
を有することを特徴とする遠隔手続き呼び出し最適化方法。

【請求項 3】

少なくとも 1 個のプログラムまたはプログラム部品を実行させる計算機上の、少なくとも 1 個の遠隔手続きを提供するサーバと該遠隔手続きを呼び出す R P C によって処理を進めるクライアントとの間で、該少なくとも 1 個の遠隔手続きの I D

L記述が提供されている際に、RPCを最適化する方法であって、
該クライアントのソースコードを解析することにより、連続して実行される可能性
が高くRPCの並びである遠隔手続き実行列を検出する処理と、
該遠隔手続き実行列を1回のRPCで実行する新たな遠隔手続きのインターフェ
ースを決定する処理と、

決定された遠隔手続きのインタフェースをIDL記述に追加登録する処理と、
を有することを特徴とする遠隔手続き呼び出し最適化方法。

【請求項4】

少なくとも1個のプログラムまたはプログラム部品を実行させる計算機上の、少
なくとも1個の遠隔手続きを提供するサーバと該遠隔手続きを呼び出すRPCに
よって処理を進めるクライアントとの間で、該少なくとも1個の遠隔手続きのIDL
記述が提供されている際に、RPCを最適化する方法であって、
該クライアントのソースコードとIDL記述とを入力する処理と、
入力されたクライアントソースコードに含まれる複数のRPCを含む第1の遠隔
手続きと、該第1の遠隔手続きのIDL記述とを作成する処理と、
作成された第1の遠隔手続きのソースコードと、作成された第1の遠隔手続きのI
DL記述と、作成された第1の遠隔手続きへの呼び出しを含む新たなクライアント
ソースコードとを出力する処理と、
を有することを特徴とする遠隔手続き呼び出し最適化方法。

【請求項5】

少なくとも1個のプログラムまたはプログラム部品を実行させる計算機上の、少
なくとも1個の遠隔手続きを提供するサーバと該遠隔手続きを呼び出すRPCに
よって処理を進めるクライアントとの間で、該少なくとも1個の遠隔手続きのIDL
記述が提供されている際に、RPCを最適化する方法であって、
該クライアントのソースコードとIDL記述とサーバのソースコードとを入力す
る処理と、
入力されたクライアントソースコードに含まれる複数のRPCを含む第1の遠隔
手続きと、該第1の遠隔手続きのIDL記述とを作成する処理と、
作成された第1の遠隔手続きのソースコードと、作成された第1の遠隔手続きのI

D L 記述と、作成された第 1 の遠隔手続きへの呼び出しを含む新たなクライアントソースコードとを出力する処理と、

を有することを特徴とする遠隔手続き呼び出し最適化方法。

【請求項 6】

少なくとも 1 個のプログラムまたはプログラム部品を実行させる計算機上の、少なくとも 1 個の遠隔手続きを提供するサーバと該遠隔手続きを呼び出す R P C によって処理を進めるクライアントとの間で、 R P C を最適化する方法であって、該クライアントのソースコードとサーバのオブジェクトとを入力する処理と、入力されたクライアントソースコードに含まれる複数の R P C を含む第 1 の遠隔手続きとを作成する処理と、

作成された第 1 の遠隔手続きのソースコードと、作成された第 1 の遠隔手続きへの呼び出しを含む新たなクライアントソースコードとを出力する処理と、
を有することを特徴とする遠隔手続き呼び出し最適化方法。

【請求項 7】

少なくとも 1 個のプログラムまたはプログラム部品を実行させる計算機上の、少なくとも 1 個の遠隔手続きを提供するサーバと該遠隔手続きを呼び出す R P C によって処理を進めるクライアントとの間で、 R P C を最適化する方法であって、該クライアントのソースコードとサーバのソースコードとを入力する処理と、入力されたクライアントソースコードに含まれる複数の R P C を含む第 1 の遠隔手続きとを作成する処理と、

作成された第 1 の遠隔手続きのソースコードと、作成された第 1 の遠隔手続きへの呼び出しを含む新たなクライアントソースコードとを出力する処理と、
を有することを特徴とする遠隔手続き呼び出し最適化方法。

【請求項 8】

少なくとも 1 個のプログラムまたはプログラム部品を実行させる計算機上の、少なくとも 1 個の遠隔手続きを提供するサーバと該遠隔手続きを呼び出す R P C によって処理を進めるクライアントとの間で、該少なくとも 1 個の遠隔手続きの I D L 記述が提供されている際に、 R P C を最適化する方法であって、
前記 I D L 記述は、少なくとも 1 個の遠隔手続きの副作用の有無と、2 個以上の遠

隔手続き間での遠隔手続きの実行順序の交換可能性または並列実行可能性の有無とを宣言する構文を有し、

該IDL記述とクライアントのソースコードとを入力する処理と、

該クライアントソースコード中のデータの流れを解析することによりまとめあげ可能なRPC列を検出する処理と、

該RPC列のインターフェースを前記IDL記述に追加する処理と、

を有することを特徴とする遠隔手続き呼び出し最適化方法。

【請求項9】

前記IDL記述は、

(1) IDL記述を格納したIDLソースファイルを読み出すことによって得られる、

(2) サーバとの通信によって得られる、

(3) サーバのファイルを参照することによって得られる、

(4) IDL記述を提供するインタフェースリポジトリとの通信によって得られる、

のいずれかである請求項3、4、5、6、7、8いずれか1項記載の遠隔手続き呼び出し最適化方法。

【請求項10】

前記クライアントソースコードから、

(1) 近接して発行される2個以上のRPCを含む処理部分を検出し、

(2) 該処理部分と同等の処理をサーバで行う第1の手続きを計算し、

(3) 該クライアントソースコードの該処理部分を該第1の手続きへの呼び出しと入れ替えて新クライアントソースコードとし、

(4) 該第1の手続きのインタフェースを前記IDL記述に追加し、

(5) 該第1の手続きの定義を前記追加サーバソースコードまたは前記新サーバソースコードに加える、

請求項4乃至7、いずれか1項記載の遠隔手続き呼び出し最適化方法。

【請求項11】

前記検出する処理は、前記クライアントソースコード中、2つ以上のRPCを発

行する基本ブロックを検索し、該基本ブロック中の第1のRPCから第2のRPCまでを含む処理部分を検出する処理を有する請求項10記載の遠隔手続き呼び出し最適化方法。

【請求項12】

前記検出する処理は、1個以上のRPCを有する基本ブロックを含むループを、前記処理部分として検出する処理を有する請求項10記載の遠隔手続き呼び出し最適化方法。

【請求項13】

前記第1の手続きを計算する処理は、データフロー解析を用いて該第1の手続きの入力引数および出力引数を得る処理を有する請求項10記載の遠隔手続き呼び出し最適化方法。

【請求項14】

前記第1の手続きを計算する処理は、前記処理部分中で同一のデータが複数の変数名から参照・変更されるエイリアス状態が発生しうるか否かを判定する処理を有する請求項10記載の遠隔手続き呼び出し最適化方法。

【請求項15】

前記エイリアス状態が発生しうるか否かを判定する処理は、変数の型に着目したデータフロー解析を用いて判定する処理を有する請求項13または14記載の遠隔手続き呼び出し最適化方法。

【請求項16】

前記エイリアスが発生しうる場合に、エイリアスがある場合には前記処理部分を実行しエイリアスがない場合には前記手続きを呼び出すという分岐処理を、前記新クライアントソースコードに追加する請求項13または14記載の遠隔手続き呼び出し最適化方法。

【請求項17】

前記第1の手続きを計算する処理は、該第1または第2の手続きへの呼び出しに必要な入力引数および出力引数の転送データ量を見積もる処理と、該見積もられた転送データ量が所定量以上である場合には、前記処理部分を該第1の手続きへの呼

び出しと入れ替えることを中止する処理とを有する請求項 10 記載の遠隔手続き呼び出し最適化方法。

【請求項 18】

前記見積もる処理は、クライアントまたはサーバに備えられた通信計測部により、プログラム実行時に R P C の回数および入力引数の転送データ量および出力引数の転送データ量を計測する処理と、計測された実行時通信情報を用いて前記転送データ量を見積もる処理とを有する請求項 17 記載の遠隔手続き呼び出し最適化方法。

【請求項 19】

前記 (1) の検出する処理は、請求項 8 に記載の副作用の有無または実行順序の交換可能性の有無に基づき、R P C の実行順序を入れ替えて、近接して発行される 2 個以上の R P C を含む処理部分を検出する処理を有する請求項 10 記載の遠隔手続き呼び出し最適化方法。

【請求項 20】

前記 (2) の第 1 の手続きを計算する処理は、請求項 8 に記載の並列実行可能性の有無に基づき、前記第 1 の手続き中に、並列実行可能な複数の遠隔手続きを複数の並列実行単位 (スレッド) によって実行するソースコードを生成する処理を有する請求項 10 記載の遠隔手続き呼び出し最適化方法。

【請求項 21】

前記サーバが、前記 1 個以上の遠隔手続きへの呼び出しを振り分けるディスパッチャとして、新たな遠隔手続きの追加を受け付ける拡張可能ディスパッチャを備える請求項 1 乃至 8、いずれか 1 項記載の遠隔手続き呼び出し最適化方法。

【請求項 22】

前記サーバが、言語を解釈実行するインタプリタまたは実行環境を内蔵し、前記クライアントが複数の R P C を実行するスクリプトをサーバに送付し、サーバが該スクリプトを該インタプリタまたは実行環境で解釈実行する請求項 1 乃至 21 いずれか 1 項記載の遠隔手続き呼び出し最適化方法。

【請求項 23】

請求項 1 乃至 8、いずれか 1 項記載の遠隔手続き呼び出し最適化方法を実行するプログラムと、

IDL 記述を、サーバ及びクライアントの RPC を行うソースコードに変換する IDL コンパイラと、

ソースコードを実行可能コードに変換するコンパイラと、

のうち 2 個以上を結合したプログラム実行方法。

【請求項 24】

請求項 1 乃至 8、いずれか 1 項記載の遠隔手続き呼び出し最適化方法を実行するプログラムを内蔵した、CORBA の IDL コンパイラまたは Sun RPC のスタブジェネレータまたは Java IDL コンパイラまたは Java RMI コンパイラ。

【請求項 25】

請求項 1 乃至 8、いずれか 1 項記載の遠隔手続き呼び出し最適化方法を実行するプログラムを、CORBA の IDL コンパイラまたは Sun RPC のスタブジェネレータまたは

Java IDL コンパイラまたは Java RMI コンパイラのうちの少なくとも 1 個の前処理用

プログラムとして用いるプログラム実行方法。

【請求項 26】

請求項 1 乃至 8、いずれか 1 項記載の遠隔手続き呼び出し最適化方法をコンピュータを用いて実行するコンピュータプログラムを格納した記憶媒体。

【請求項 27】

請求項 1 乃至 8 のいずれか 1 項記載の遠隔手続き呼び出し最適化方法を用いて実行するプログラムに、クライアント毎のクライアントソースコードと IDL ソースコードとを入力する処理と、

該プログラムにより最適化された遠隔手続き呼び出しに対応してそれぞれクライアント毎の新たなクライアントソースコードと新たな IDL ソースコードとサーバ毎の追加サーバソースコードとを該プログラムから出力する処理と、

該新たな IDL ソースコードを IDL コンパイラに入力してクライアントスタブ

、 R P Cヘッダファイル及びサーバスタブを出力する処理と、
該クライアントスタブ、 R P Cヘッダファイル及びサーバスタブと、前記クライアント毎の新たなクライアントソースコードとサーバ毎の追加サーバソースコードとをコンパイル及びリンクして、クライアントオブジェクトと追加サーバオブジェクトとを出力する処理と、
該追加サーバオブジェクトとサーバオブジェクトとを動的にリンクして新サーバオブジェクトを出力する処理と、
該クライアントオブジェクトを第 1 の計算機により実行する処理と、
該クライアントオブジェクトを第 1 の計算機により実行するために該クライアントオブジェクトから R P Cを送信する処理と、
該 R P Cを受信した前記新サーバオブジェクトを第 2 の計算機により実行する処理と、
を有することを特徴とするプログラム実行方法。

【請求項 2 8】

前記送信する処理は、実行時通信情報を計測するために前記第 1 の計算機に設けられた第 1 の通信計測部から第 2 の計算機に設けられた第 2 の通信計測部に R P Cを送信する処理を有する請求項 2 7 記載のプログラム実行方法。

【請求項 2 9】

前記送信する処理は、前記第 1 及び第 2 の通信計測部のすくなくとも一方で計測された実行時通信情報を、前記遠隔手続き呼び出し最適化方法を実行するプログラムに送信する処理を有する請求項 2 8 記載のプログラム実行方法。

【請求項 3 0】

請求項 1 乃至 8 のいずれか 1 項記載の遠隔手続き呼び出し最適化方法を用いて実行するプログラムに、クライアント毎のクライアントソースコードと I D Lソースコードとサーバ毎のサーバソースコードとを入力する処理と、
該プログラムにより最適化された遠隔手続き呼び出しに対応してそれぞれクライアント毎の新たなクライアントソースコードと新たな I D Lソースコードとサーバ毎の新たなサーバソースコードとを該プログラムから出力する処理と、
該新たな I D Lソースコードを I D Lコンパイラに入力してクライアントスタブ

、 R P Cヘッダファイル及びサーバスタブを出力する処理と、
該クライアントスタブ、 R P Cヘッダファイル及びサーバスタブと、前記クライアント毎の新たなクライアントソースコードとサーバ毎の新たなサーバソースコードとをコンパイル及びリンクして、クライアントオブジェクトとサーバオブジェクトとを出力する処理と、
を有することを特徴とするプログラム実行方法。

【請求項 3 1】

請求項 1 乃至 8 のいずれか 1 項記載の遠隔手続き呼び出し最適化方法を用いて実行するプログラムに、クライアント毎のクライアントソースコードとサーバ毎のサーバクラスコードとを入力する処理と、
該プログラムにより最適化された遠隔手続き呼び出しに対応してそれぞれクライアント毎の新たなクライアントソースコードとサーバ毎の追加サーバソースコードとを該プログラムから出力する処理と、
出力されたクライアント毎の新たなクライアントソースコードとサーバ毎の追加サーバソースコードとをコンパイルしてそれぞれ J a v a 仮想マシン（以下、J V M）で実行可能なクライアントクラスコード及び追加サーバクラスコードとを出力する処理と、
出力された追加サーバクラスコードを J a v a 言語の遠隔手続き呼び出し（以下、R M I）コンパイラによってコンパイルしてクライアントスタブとサーバスタブとを出力する処理と、
出力されたクライアントクラスコードとクライアントスタブとを第 1 の J V M で実行する処理と、
出力された追加サーバクラスコードとサーバスタブとを第 2 の J V M で実行する処理と、
を有することを特徴とするプログラム実行方法。

【請求項 3 2】

請求項 1 乃至 8 のいずれか 1 項記載の遠隔手続き呼び出し最適化方法を用いて実行するプログラムに、クライアント毎のクライアントソースコードとサーバ毎のサーバソースコードとを入力する処理と、

該プログラムにより最適化された遠隔手続き呼び出しに対応してそれぞれクライアント毎の新たなクライアントソースコードとサーバ毎の新たなサーバソースコードとを該プログラムから出力する処理と、

出力されたクライアント毎の新たなクライアントソースコードとサーバ毎の新たなサーバソースコードとをコンパイルしてそれぞれJ a v a仮想マシン（以下、J V M）で実行可能なクライアントクラスコード及びサーバクラスコードとを出力する処理と、

出力された追加サーバクラスコードをJ a v a言語の遠隔手続き呼び出し（以下、R M I）コンパイラによってコンパイルしてクライアントスタブとサーバスタブとを出力する処理と、

出力されたクライアントクラスコードとクライアントスタブとを第1のJ V Mで実行する処理と、

出力された追加サーバクラスコードとサーバスタブとを第2のJ V Mで実行する処理と、

を有することを特徴とするプログラム実行方法。

要約書

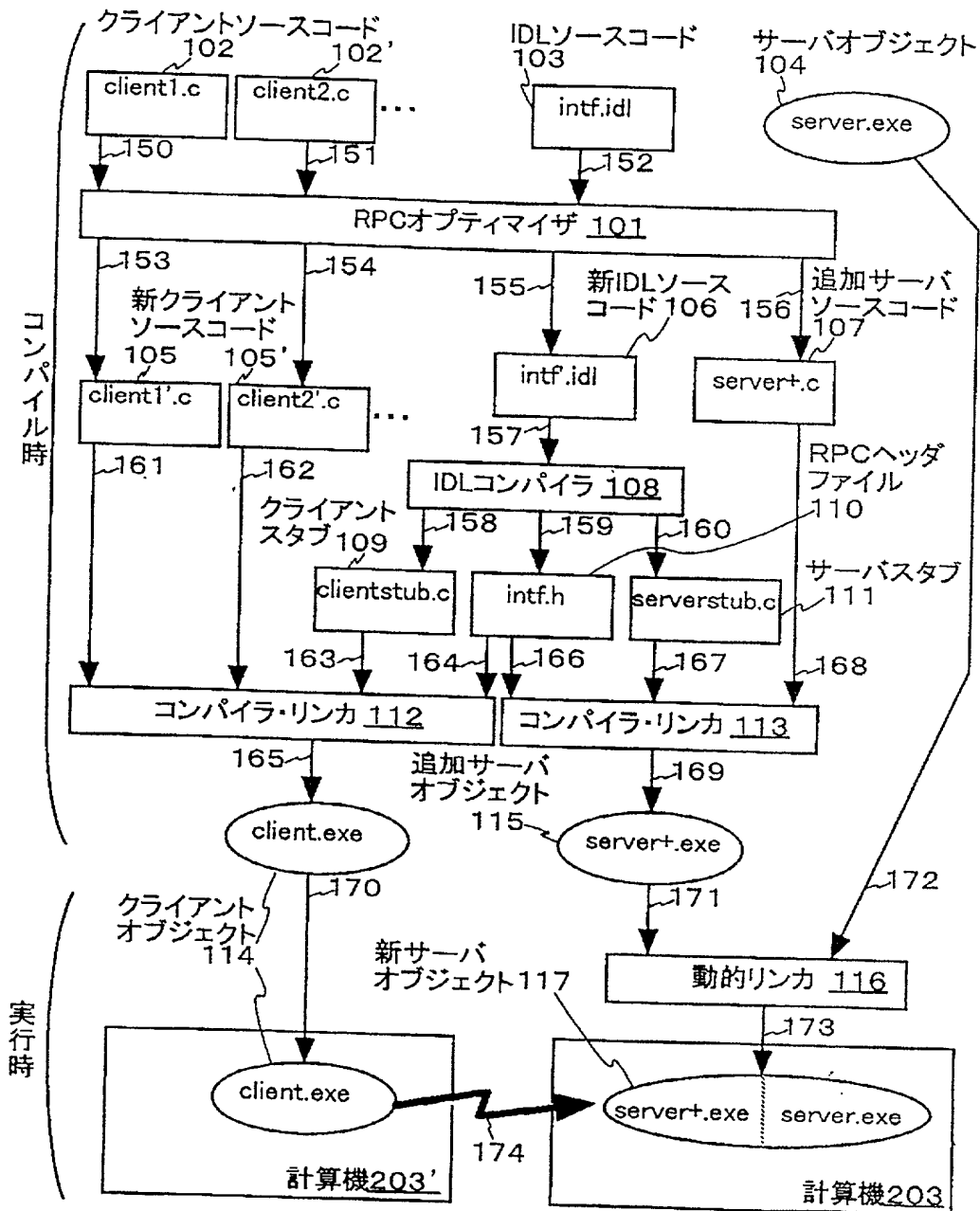
(ABSTRACT OF THE DISCLOSURE)

クライアントとサーバの間の一連の遠隔手続き呼び出しを高速に実行するため、IDLソースコード(103)と、クライアントソースコード(102、102'、…)とを入力に用いるRPCオプティマイザ(101)により、クライアントが行う一連の遠隔手続き群を新たな遠隔手続きとして、追加サーバソースコード(107)に定義するとともに、該遠隔手続きのインタフェースを追加した新IDLソースコード(106)と、該遠隔手続きを利用するよう変更した新クライアントソースコード(105、105'、…)を出力することにより、複数回の遠隔手続きによる通信が1回に低減され、クライアントとサーバ間の高速通信が可能となる。

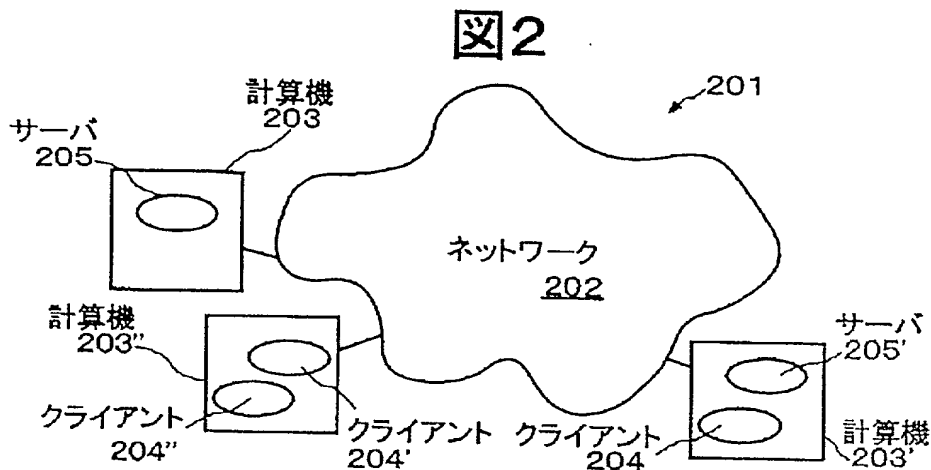
【書類名】 図面

【図 1】

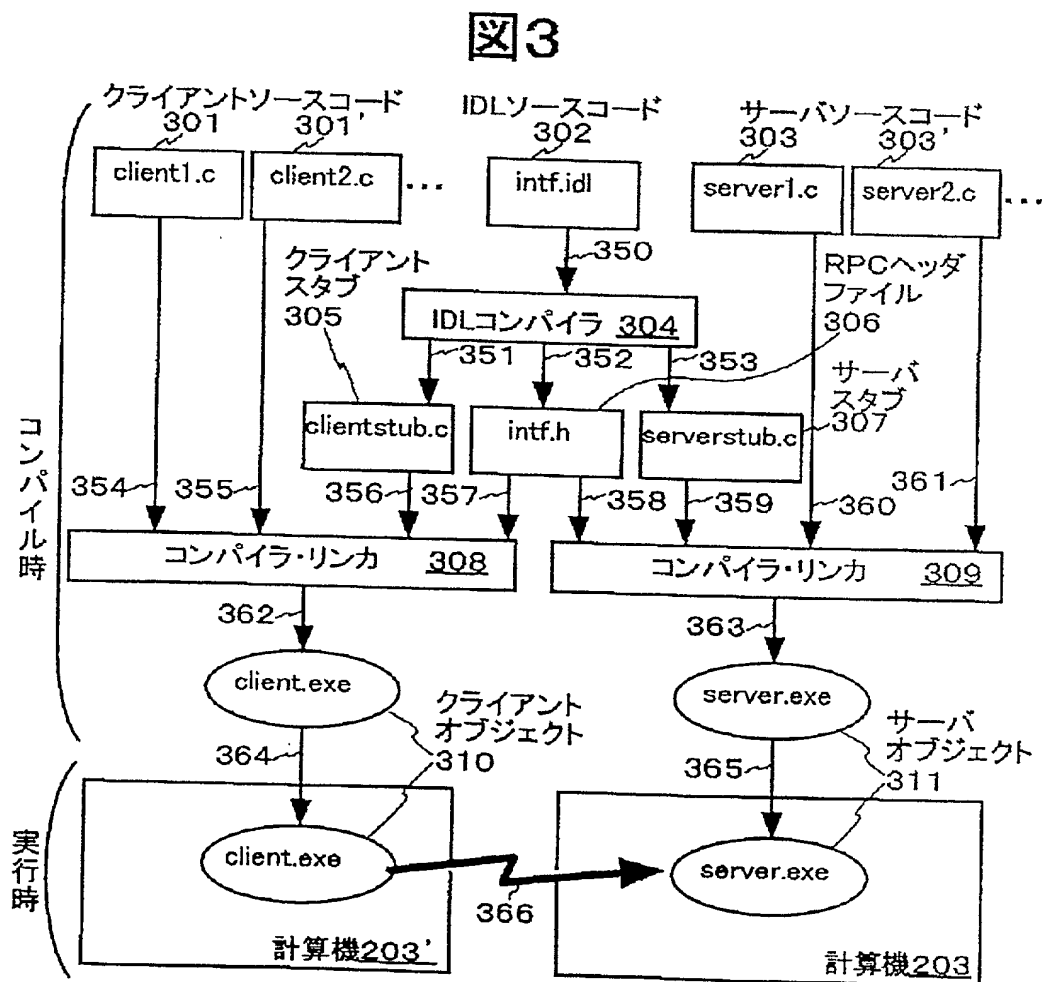
図 1



【図 2】

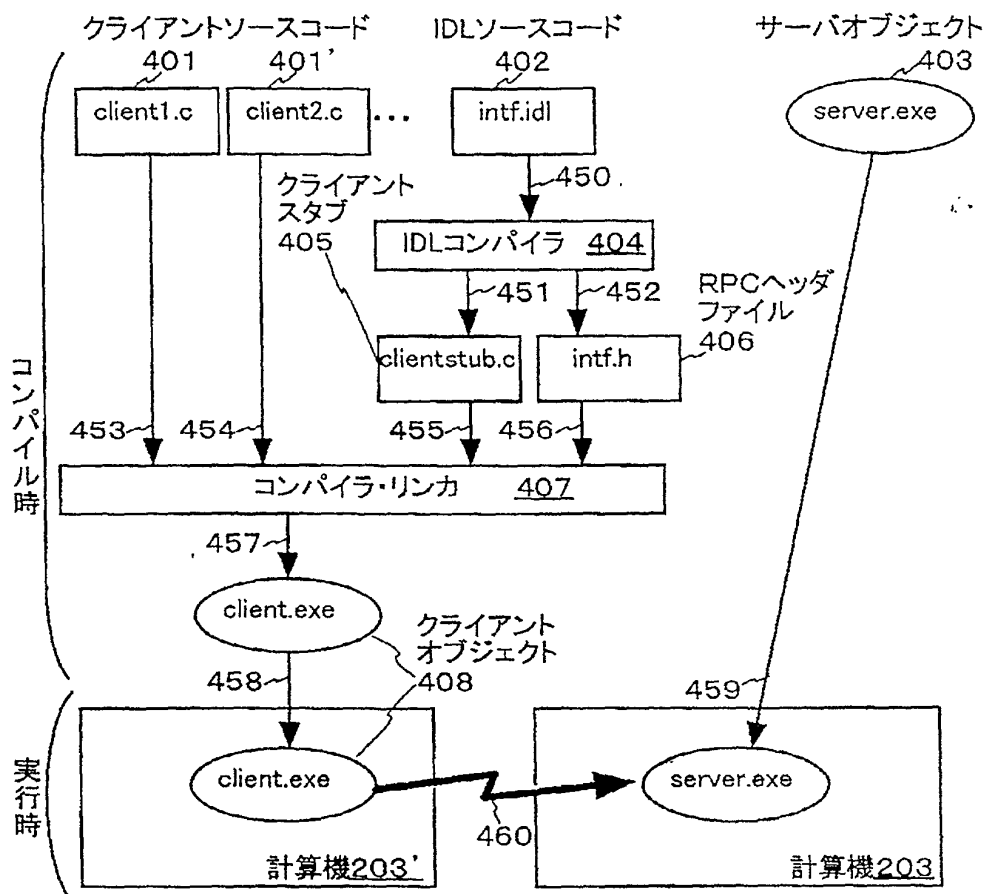


【図 3】



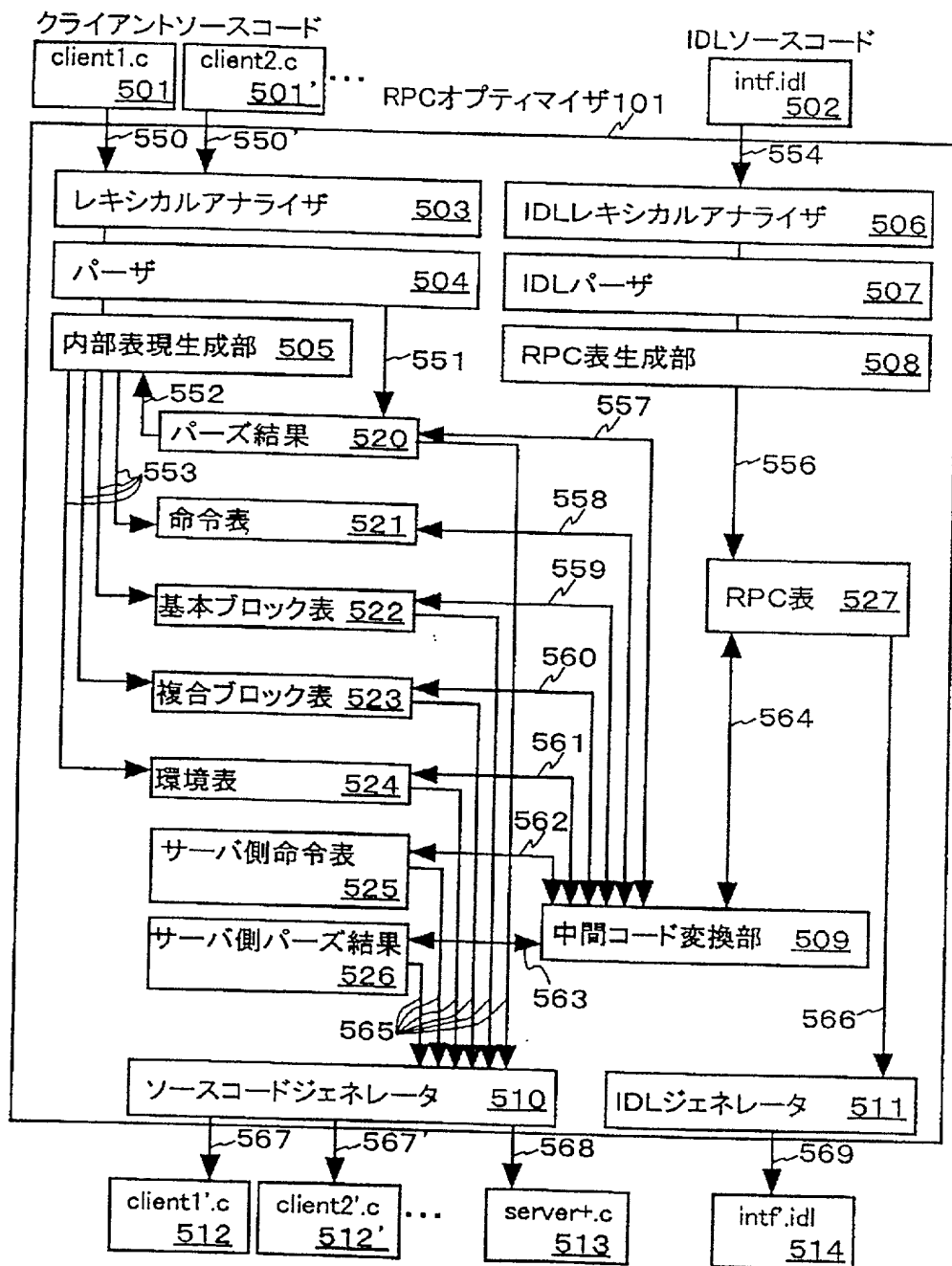
【図4】

図4



【図5】

図5



【図 6】

図6

命令表 600

命令ID 602	ターゲット 603	命令 604	オペランドA 605	オペランドB 606
命令要素 601 ✓				
:				

基本ブロック表 610

基本ブロックID 612	開始命令ID 613	終端命令ID 614
次基本ブロック 615	前基本ブロック 616	環境ID 617
DGEN変数表 618	DKILL変数表 619	DIN変数表 620
DOUT変数表 621	LIN変数表 622	LOUT変数表 623
LUSE変数表 624	LDEF変数表 625	
基本ブロック要素 611 ✓		
:		

複合ブロック表 630

複合ブロックID 632	開始基本ブロックID 633	終端基本ブロックID 634	環境ID 635
複合ブロック要素 631 ✓			
:			

環境表 640

環境ID 641	親環境ID 642	属性 643
環境内変数表 644		

RPC表 650

RPC名 652	IN引数表 653	OUT引数表 654	属性 655
RPC表要素651✓			
⋮			
型名 656	型情報 657		
型宣言要素658✓			
⋮			

変数表 660

変数名 662	型 663	属性 664
変数表要素 661 ✓		
:		

$\frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx$

[illegible][illegible][illegible]

$\frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx$

$\frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx$

$\frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx$

[illegible][illegible]

$\frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx$

[illegible]

$\frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx$

[illegible][illegible][illegible]

$\frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} f(x) e^{-x^2} dx$

[illegible][illegible][illegible]

【図 8】

図 8

intf.h

```
801 #include "Object.h"
802 class MyServer: public Object {
803     int func1(int i);
804     void func2(long& key, char* value);
805 }
```

800

clientstub.c

```
851 #include "intf.h"
852 int MyServer::func1(int i)
853 {
854     Buffer buf = new Buffer();
855     int rval;
856     buf.packint(i);
857     call("func1", buf);
858     buf.unpackint(&rval);
859     delete buf;
860     return rval;
861 }
862 void MyServer::func2(long& key, char* value)
863 {
864     Buffer buf = new Buffer();
865     buf.packlong(key);
866     buf.packString(value);
867     call("func2", buf);
868     buf.unpacklong(&key);
869     delete buf;
870 }
```

850

66260"6636460

【図 9】

図 9

```
serverstub.c 900
901 #include "intf.h"
902 void MyServer::loop()
903 {
904     while (1) {
905         Buffer buf;
906         Client client;
907         receive(&client, &buf);
908         if (buf.method.equals("func1")) {
909             int i, rval;
910             buf.unpackint(&i);
911             rval = func1(i);
912             buf.packint(rval);
913         } else if (buf.method.equals("func2")) {
914             long key;
915             char* value;
916             buf.unpacklong(&key);
917             buf.unpackString(&value);
918             func2(key, value);
919             buf.packlong(key);
920         } else {
921             send(client, "error");
922             continue;
923         }
924         send(client, buf);
925         delete buf;
926         delete client;
927     }
928 }
```

図10

intf.idl

```
1001 interface MyServer {  
1002     int func1(in int i);  
1003     void func2(inout long key, in String value);  
1004     void func3(inout int count);  
1005     void func4(in int i);  
1006 };
```

1000

client1.c

```
1011 #include "intf.h"  
1012 main()  
1013 {  
1014     MyServer server = lookupDirectory("MyServer");  
1015     int count = 0;  
1016     server.func3(count);  
1017     printf("count=%d\n", count);  
1018     server.func4(j);  
1019 }
```

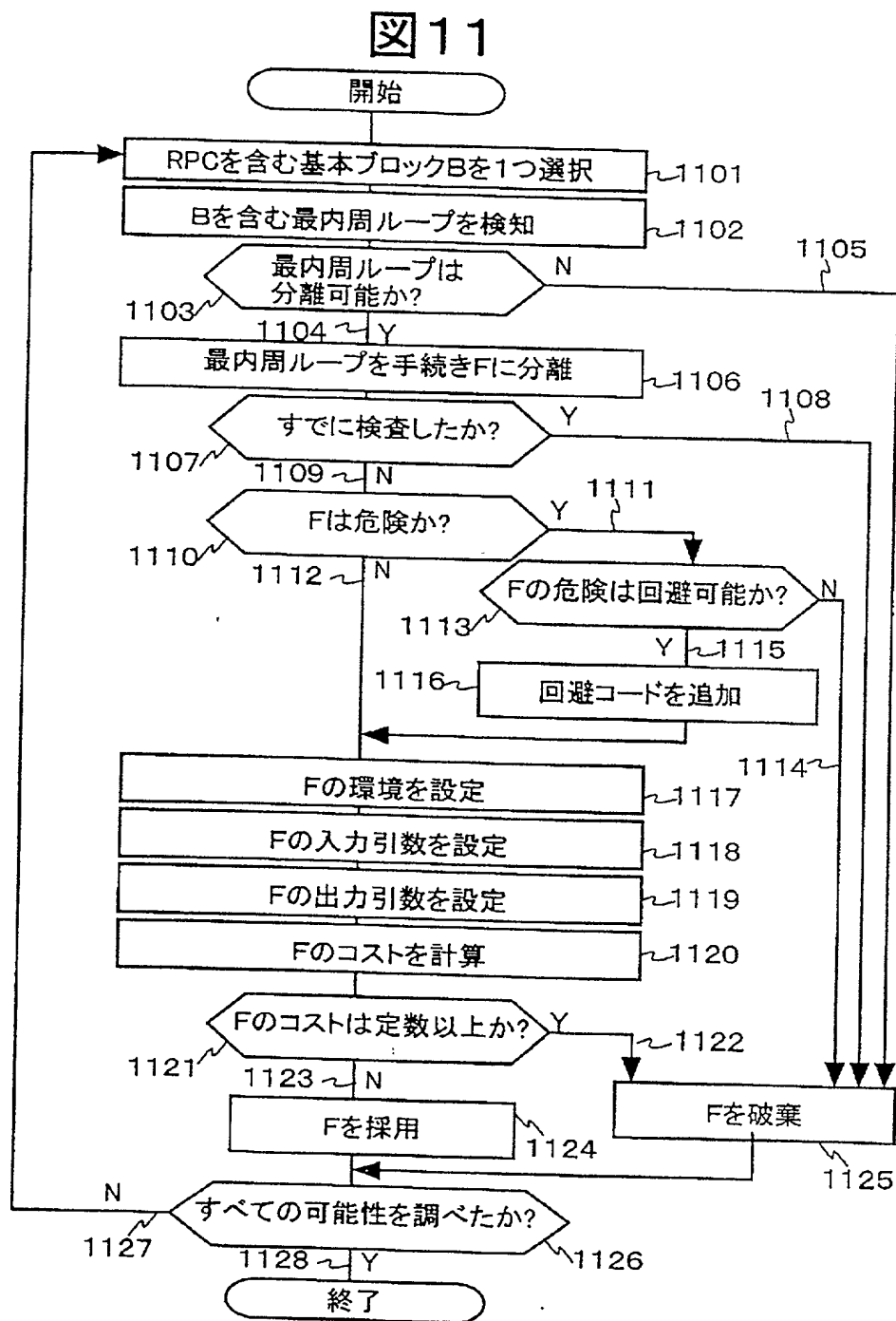
1010

server1.c

```
1031 #include "intf.h"  
1032 void MyServer::func3(int& count)  
1033 {  
1034     for (int i = 0; i < 100; i++)  
1035         count += server.func1(i);  
1036 }  
  
1037 void MyServer::func4(int count)  
1038 {  
1039     server.func2(100, "hello world");  
1040     server.func1(count);  
1041 }
```

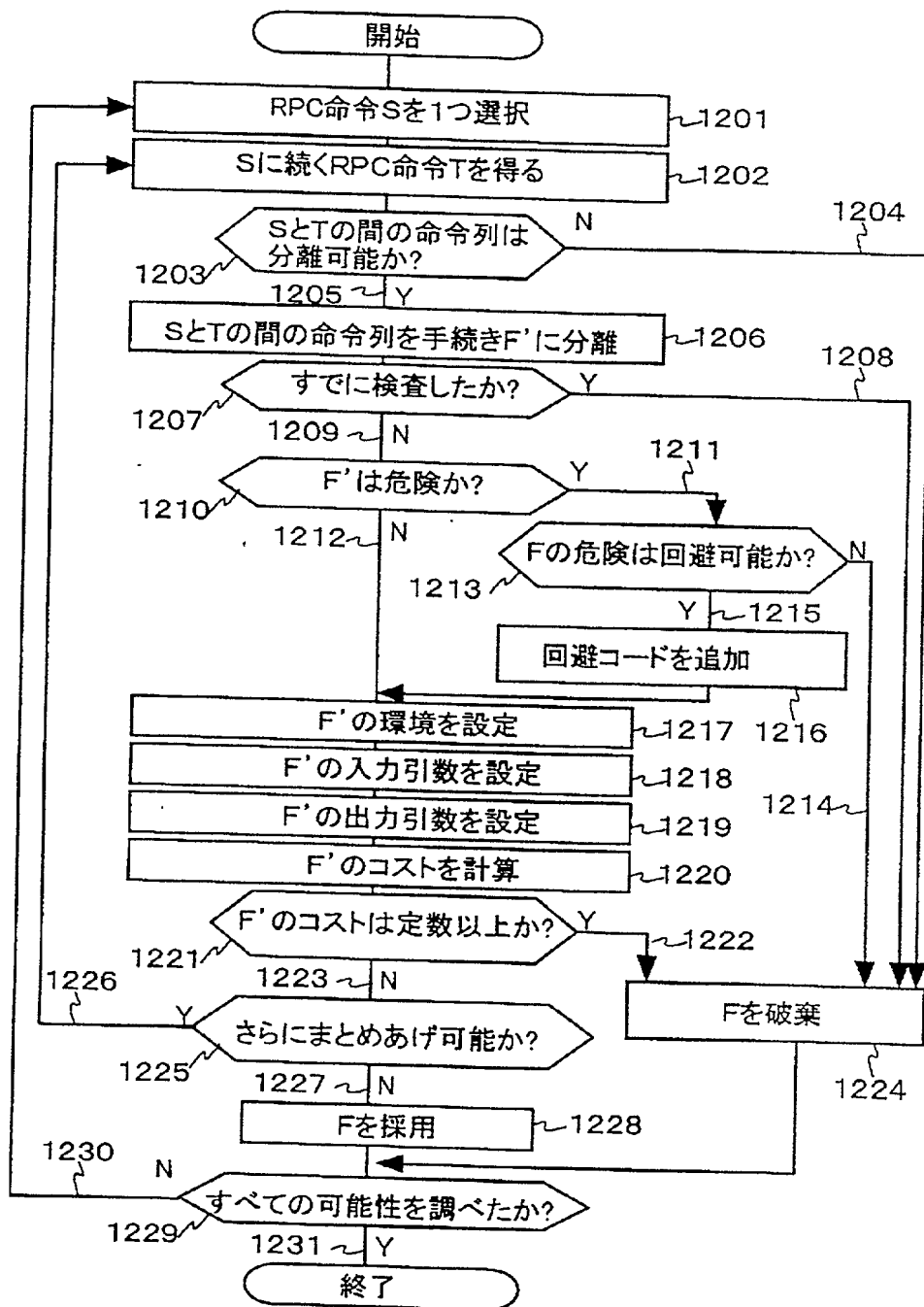
1030

【図11】

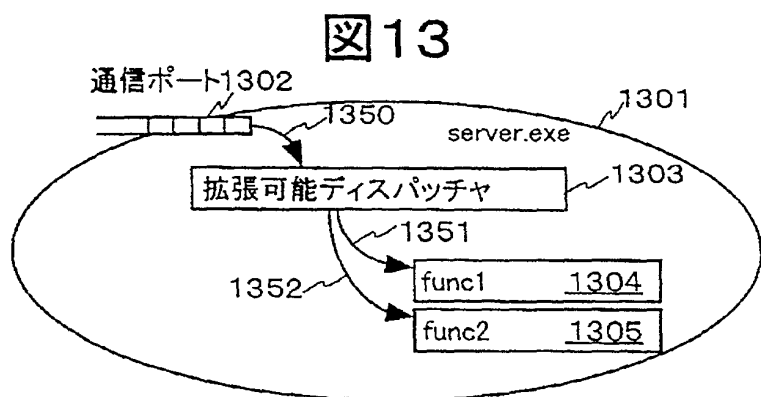


【図12】

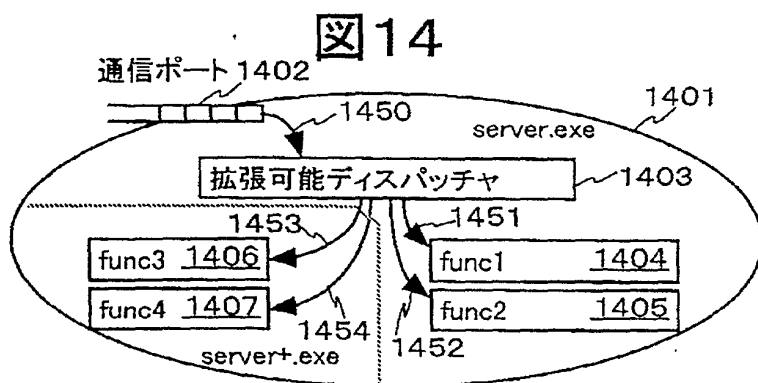
図12



【図13】

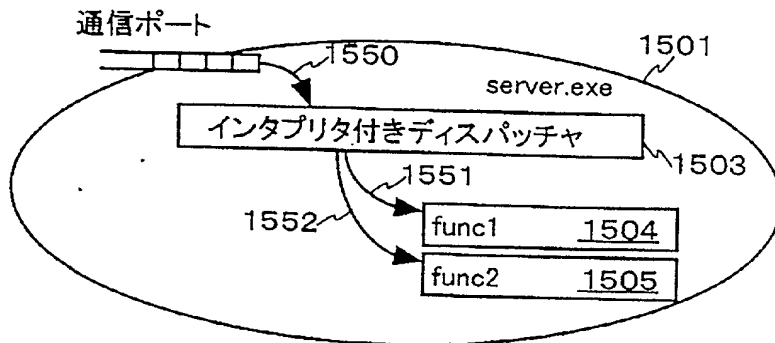


【図14】



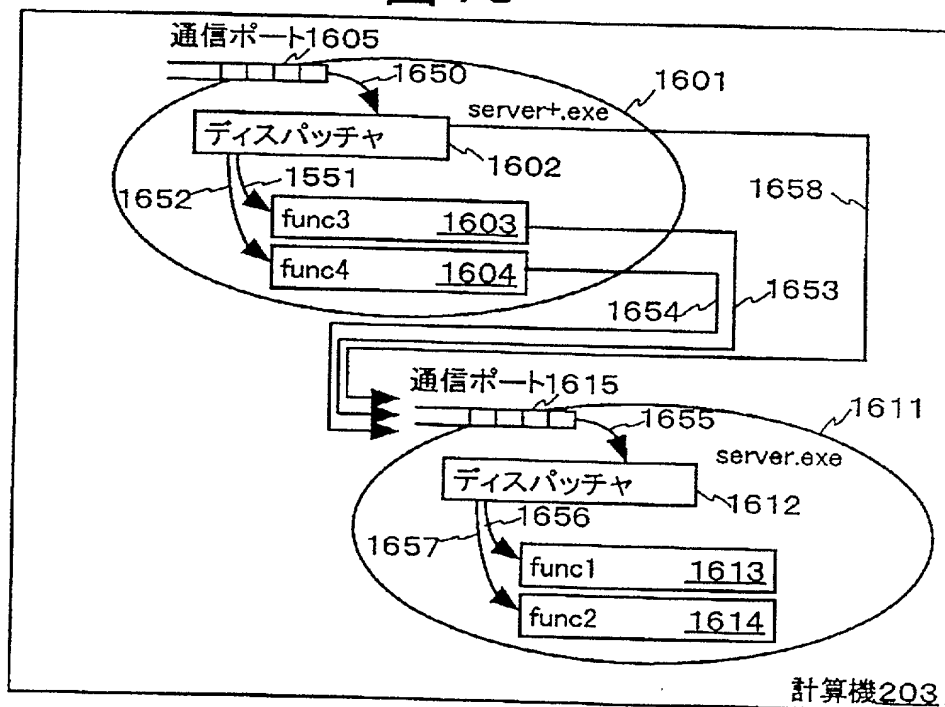
【図15】

図15



【図16】

図16



【図17】

図17

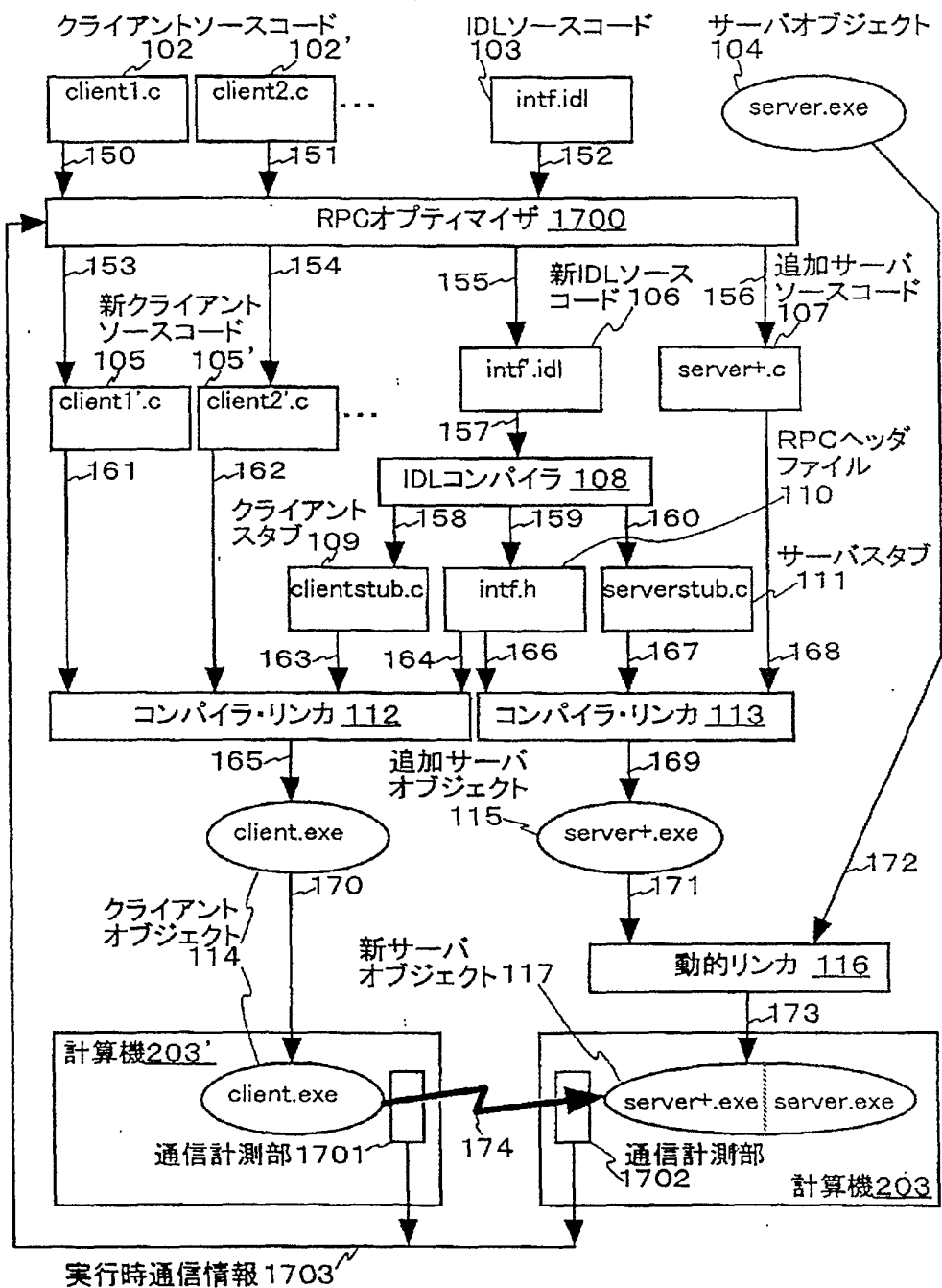


图 18

```

    extended intf.idl
1801 interface MyServer {
1802     int func1(in int i) const;
1803     void func2(inout long key, in String value);
1804     int func3(void);

1805     commutative { func2, func3 };

1806     parallel { func1, func2, func3 };
1807 };
1800

server.c
1821 #include "intf.h"
1822 #include "thread.h"

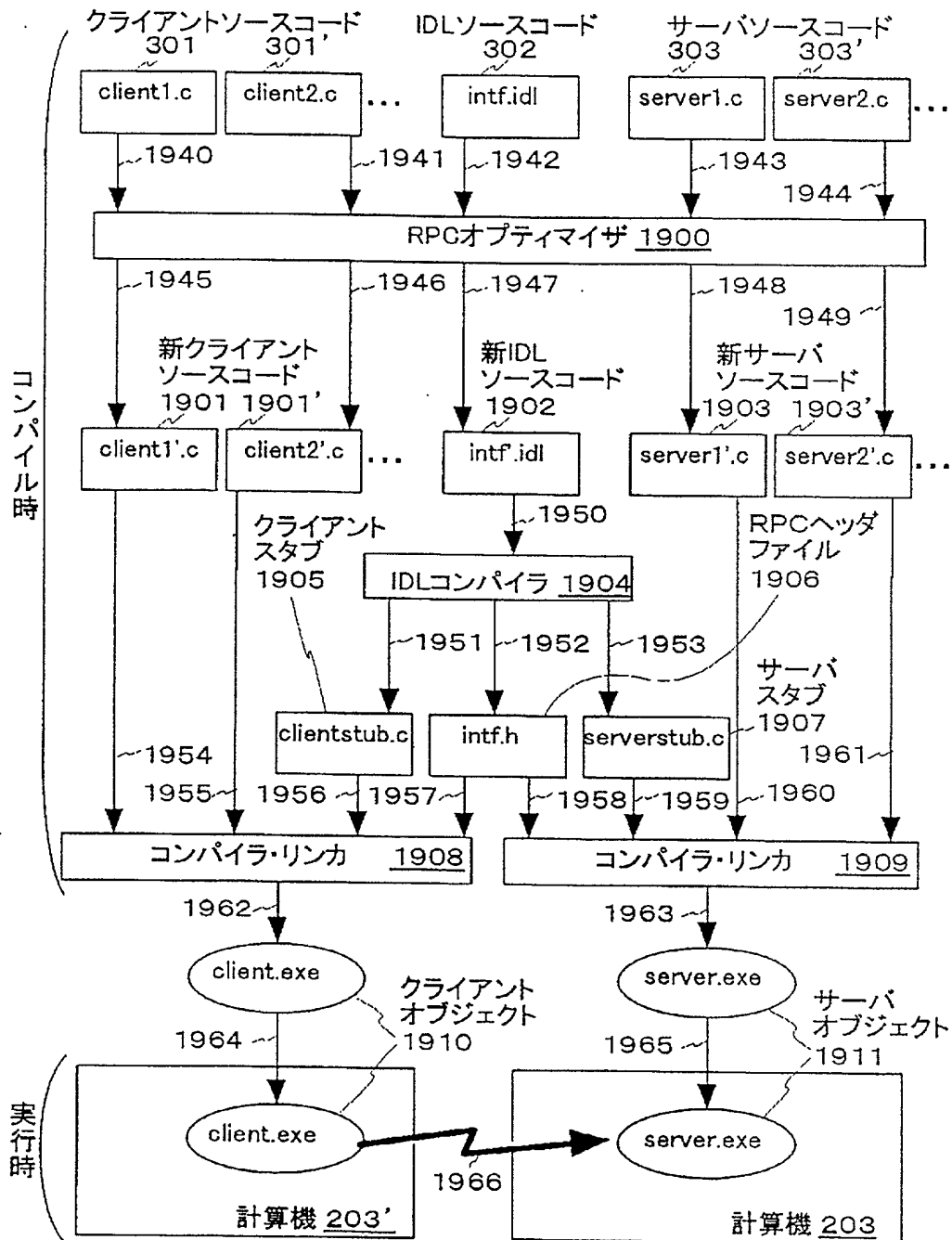
1823 void MyServer::func3(int& count)
1824 {
1825     List<Thread> allThreads;
1826     Thread t;
1827     void *rval;
1828     for (int i = 0; i < 100; i++) {
1829         create_thread(&t, server.func1, 1, i);
1830         allThreads.add(t);
1831     }
1832     for ( ; (t = allThreads.next()) != NULL_THREAD; ) {
1833         join_thread(t, &rval);
1834         count += *(int *)rval;
1835     }
1836 }

1837 void MyServer::func4(int count)
1838 {
1839     List<Thread> allThreads;
1840     Thread t;
1841     create_thread(&t, server.func2, 2, 100, "hello world");
1842     allThreads.add(t);
1843     create_thread(&t, server.func1, 1, count);
1844     allThreads.add(t);
1845     for ( ; (t = allThreads.next()) != NULL_THREAD; )
1846         join_thread(t, NULL);
1847 }
1820

```

【図19】

図19



【図20】

図20

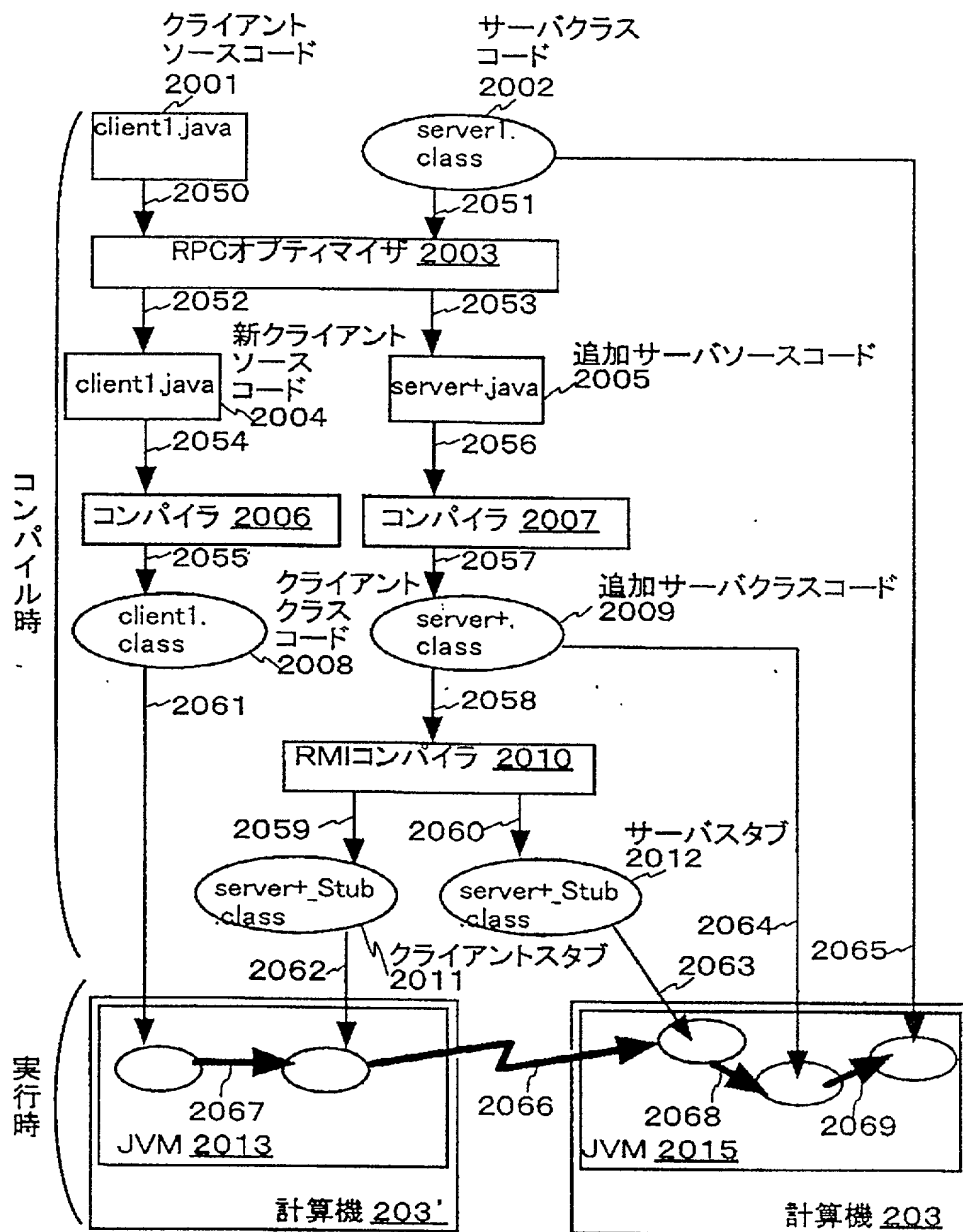
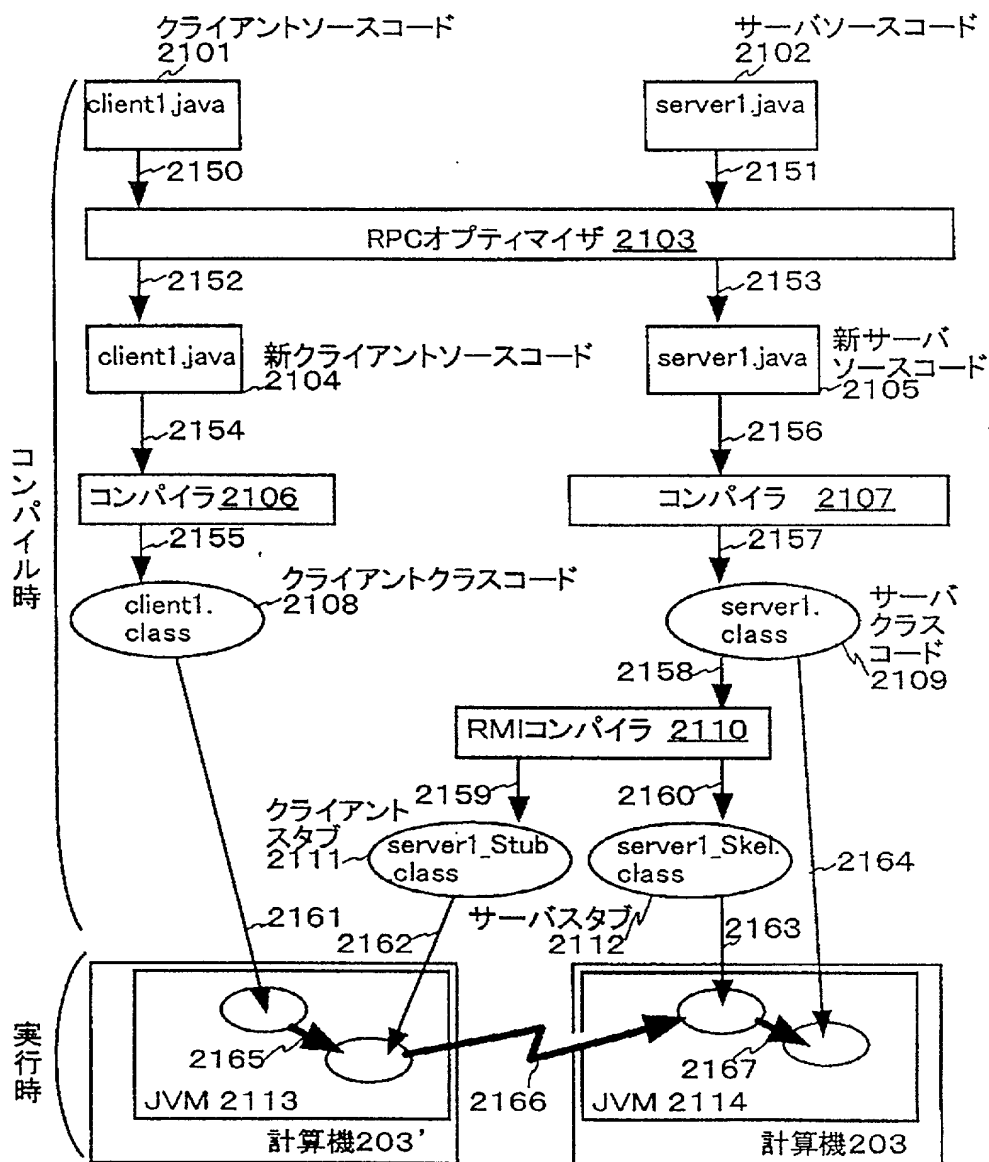


図21



Please type a plus sign (+) inside this box → ☐

PTO/SB/122 (11-96)
Approved for use through 6/30/99. OMB 0651-0035
Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

CHANGE OF CORRESPONDENCE ADDRESS *Application*

Address to:
Assistant Commissioner for Patents
Washington, D.C. 20231

Application Number

Filing Date

September 27, 1999

First Named Inventor

S. INOHARA, et al

Group Art Unit

Examiner Name

Attorney Docket Number

520.37631X00

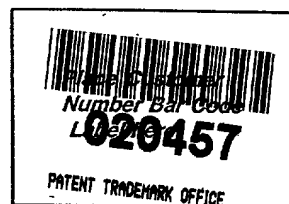
Please change the Correspondence Address for the above-identified application to:



Customer Number

020457

Type Customer Number here



OR



Firm or
Individual Name

Address

Address

City

State

ZIP

Country

Telephone

Fax

This form cannot be used to change the data associated with a Customer Number. To change the data associated with an existing Customer Number use "Request for Customer Number Data Change" (PTO/SB/124).

I am the :



Applicant.



Assignee of record of the entire interest.
Certificate under 37 CFR 3.73(b) is enclosed.



Attorney or agent of record .

Typed or
Printed Name

Carl I. Brundidge

Registration NO. 29,621

Signature

Date

September 27, 1999

Burden Hour Statement: This form is estimated to take 0.2 hours to complete. Time will vary depending upon the needs of the individual case. Any comments on the amount of time you are required to complete this form should be sent to the Chief Information Officer, Patent and Trademark Office, Washington, DC 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Assistant Commissioner for Patents, Washington, DC 20231.